Programmazione ed Algoritmica

Appunti del corso di Laurea in Informatica

Diego Stefanini

cc. Davide Paolocchi

Anno Accademico 2025–2026

Università degli Studi di Pisa

Indice

1 Introduzione

1.1 Cos'è l'Informatica

> **Definizione 1.1 – Informatica.** Lo studio sistematico degli **algoritmi** che descrivono e trasformano le informazioni: la loro teoria, analisi, progettazione, efficienza, implementazione e applicazione. (ACM – Association for Computing Machinery)

L'informatica non è semplicemente lo studio dei computer, ma lo studio dei *processi algoritmici*: come rappresentare l'informazione, come trasformarla e come farlo in modo efficiente. Il calcolatore è lo strumento che permette di eseguire tali processi in modo rapido e affidabile.

1.2 Algoritmo

> **Definizione 1.2 – Algoritmo.** Una sequenza **finita** di **passi** (istruzioni) **univocamente determinati** che, se eseguiti da un **esecutore**, portano alla risoluzione di un **problema**.

Le parole chiave di questa definizione meritano un approfondimento:
- **Passi**: operazioni elementari, cioè istruzioni la cui esecuzione è ben definita e non ulteriormente scomponibile dal punto di vista dell'esecutore.
- **Esecutore**: l'entità (umano o macchina) che esegue i passi dell'algoritmo. Nel nostro caso l'esecutore è un calcolatore.
- **Problema**: la domanda a cui l'algoritmo deve rispondere, specificata in modo preciso tramite input e output.

Le tre componenti fondamentali di un algoritmo sono dunque:
1. il **problema** da risolvere,
2. il **procedimento** (la sequenza di passi) da seguire,
3. l'**esecutore** che interpreta ed esegue le istruzioni.

> **Nota.** La descrizione dell'algoritmo deve essere comprensibile dall'esecutore: un algoritmo corretto ma scritto in un linguaggio che l'esecutore non comprende risulta inutile.

1.2.1 Proprietà degli algoritmi

Un algoritmo deve soddisfare le seguenti proprietà:

- **Finitezza**: l'algoritmo è composto da un numero finito di passi e la sua esecuzione deve terminare in tempo finito per ogni input valido.
- **Non ambiguità**: ogni passo è definito in modo preciso, senza possibilità di interpretazioni diverse. L'esecutore non deve mai trovarsi nella condizione di non sapere cosa fare.
- **Determinismo**: dato lo stato corrente dell'esecuzione, il passo successivo è univocamente determinato. A parità di input, l'algoritmo produce sempre lo stesso output.
- **Generalità**: l'algoritmo risolve un'intera *classe* di problemi, non una singola istanza specifica.

Esempio 1.1 – Algoritmo per il massimo di un vettore. Problema: dato un array $A[1..n]$ di n interi, determinare il valore massimo.

- **Input**: array A di n interi
- **Output**: il valore m tale che $m = \max\{A[i] : 1 \leq i \leq n\}$

Algoritmo (in linguaggio naturale):
1. Assumi il primo elemento come massimo corrente.
2. Per ciascun elemento successivo, confrontalo con il massimo corrente: se è maggiore, aggiorna il massimo corrente.
3. Al termine, restituisci il massimo corrente.

In MAO:

Massimo di un array

```
int max(int[] A, int n){
    int m = A[1];
    int i = 2;
    while(i <= n){
        if(A[i] > m){
            m := A[i];
        }
        i := i + 1;
    }
    return m;
}
```

1.3 Programma e programmazione

Definizione 1.3 – Programma. La formulazione di un algoritmo in un **linguaggio di programmazione**. Un programma specifica al calcolatore quali operazioni eseguire, in quale ordine, su quali dati e sotto quali condizioni.

Definizione 1.4 – Programmazione. L'attività di progettare e scrivere programmi, ossia tradurre un algoritmo in una sequenza di istruzioni eseguibili da un calcolatore.

1.3.1 Differenza tra algoritmo e programma

Un algoritmo è un concetto *astratto*, indipendente dal linguaggio in cui viene espresso. Un programma è la sua realizzazione *concreta* in un linguaggio specifico. Lo stesso algoritmo può essere implementato in linguaggi diversi, ottenendo programmi differenti ma semanticamente equivalenti.

Aspetto	Algoritmo	Programma
Livello	Astratto	Concreto
Linguaggio	Naturale / pseudocodice	Linguaggio di programmazione
Esecutore	Umano o macchina	Solo macchina
Dettagli implementativi	Possono essere omessi	Tutti specificati

Tabella 1: Confronto tra algoritmo e programma

1.4 Computer e problem solving

> **Definizione 1.5 – Computer.** Un **calcolatore** è una macchina in grado di eseguire operazioni elementari con grande **rapidità** e **precisione**, seguendo le istruzioni di un programma.

Un computer riceve in **input** un programma (testo) e un insieme di dati, e produce in **output** il risultato dell'esecuzione del programma sui dati forniti. A differenza di altre macchine automatiche (ad esempio una lavatrice, il cui comportamento è fisso), un computer è **programmabile**: il compito svolto dipende interamente dal programma caricato.

1.4.1 Problem solving

> **Definizione 1.6 – Problem solving.** Attività sistematica finalizzata all'analisi e alla risoluzione di *problemi computazionali*, ovvero problemi formulabili in termini di input, output e di una relazione tra essi.

Il processo di problem solving si articola nelle seguenti fasi:

1. **Specifica**: definizione precisa del problema, stabilendo chiaramente quali sono gli input ammissibili e qual è l'output atteso.
2. **Progettazione**: ideazione di un algoritmo che risolve il problema, ragionando sulla strategia risolutiva.
3. **Analisi**: studio delle proprietà dell'algoritmo, in particolare la sua *correttezza* (produce l'output corretto per ogni input valido) e la sua *efficienza* (quanto tempo e quanta memoria richiede).
4. **Codifica**: traduzione dell'algoritmo in un linguaggio di programmazione, ottenendo un programma.
5. **Testing e debugging**: verifica sperimentale della correttezza del programma su input di prova e correzione degli eventuali errori.
6. **Esecuzione**: esecuzione del programma sul calcolatore.

> **Nota – Problemi irrisolvibili.** Non tutti i problemi computazionali ammettono una soluzione algoritmica. Esistono problemi per i quali si può dimostrare che *non esiste alcun algoritmo* in grado di risolverli. Un esempio celebre è il **Problema della fermata** (*Halting Problem*): dato un programma P e un input x, stabilire se P termina quando eseguito su x. Alan Turing dimostrò nel 1936 che non esiste alcun algoritmo in grado di rispondere correttamente per ogni coppia (P, x).

1.5 Problemi computazionali

> **Definizione 1.7 – Problema computazionale.** Un problema formulato in modo preciso di cui si cerca una soluzione algoritmica. Un problema computazionale è definito da tre elementi:
> - **Input**: l'insieme dei dati in ingresso, cioè la descrizione di tutte le possibili *istanze* del problema.
> - **Output**: il risultato atteso.
> - **Relazione input-output**: il vincolo che lega l'output all'input, specificando quale output è corretto per ciascun input.

Definizione 1.8 – Istanza di un problema. Un'istanza è un input specifico, cioè un particolare elemento dell'insieme degli input ammissibili. Un algoritmo che risolve un problema deve produrre l'output corretto per *ogni* istanza.

Esempio 1.2 – Problema dell'ordinamento.
- **Input**: una sequenza di n numeri $\langle a_1, a_2, ..., a_n \rangle$
- **Output**: una permutazione $\langle a'_1, a'_2, ..., a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq ... \leq a'_n$

Ad esempio, data l'istanza $\langle 5, 2, 8, 1 \rangle$, l'output corretto è $\langle 1, 2, 5, 8 \rangle$.

Esempio 1.3 – Problema della ricerca.
- **Input**: una sequenza di n numeri $\langle a_1, a_2, ..., a_n \rangle$ e un valore k
- **Output**: un indice i tale che $a_i = k$, oppure -1 se k non compare nella sequenza

Ad esempio, data l'istanza $\langle 3, 7, 1, 9 \rangle$ con $k = 7$, l'output corretto è $i = 2$.

1.6 Correttezza e terminazione

Dato un problema computazionale, un algoritmo che intende risolverlo deve soddisfare due requisiti fondamentali.

Definizione 1.9 – Correttezza. Un algoritmo si dice **corretto** rispetto a un problema computazionale se, per ogni istanza valida dell'input, produce l'output corretto (cioè l'output che soddisfa la relazione input-output del problema).

Definizione 1.10 – Terminazione. Un algoritmo si dice **terminante** se, per ogni istanza valida dell'input, la sua esecuzione si conclude in un numero finito di passi.

Nota. Un algoritmo che produce risultati corretti ma non termina su certi input non è considerato un algoritmo valido. Analogamente, un algoritmo che termina sempre ma produce risultati errati non è utile. Sono necessarie *entrambe* le proprietà: correttezza e terminazione.

1.7 Linguaggi di Programmazione

Abbiamo visto che un algoritmo ed un programma sono concetti distinti: il primo è una descrizione astratta di un procedimento risolutivo, il secondo è la sua formulazione concreta in un linguaggio comprensibile da una macchina. Il **linguaggio di programmazione** è lo strumento che consente questa traduzione.

Definizione 1.11 – Linguaggio di programmazione. Un linguaggio di programmazione è un linguaggio formale dotato di una **sintassi** rigorosa e di una **semantica** precisa, progettato per esprimere algoritmi in una forma eseguibile da un calcolatore.

I linguaggi di programmazione condividono alcune caratteristiche fondamentali:
- sono **formali**: ogni costrutto ha una definizione precisa e non ambigua;

- sono **eseguibili**: un programma scritto correttamente può essere eseguito da una macchina;
- sono **espressivi**: permettono di descrivere algoritmi che risolvono problemi computazionali.

> **Nota.** Un linguaggio di programmazione funge da **ponte** tra il ragionamento umano e l'esecuzione automatica. Esistono centinaia di linguaggi (C, Python, Java, JavaScript, Haskell, …), ciascuno con caratteristiche proprie, ma tutti fondati sugli stessi concetti di base che studieremo in questo corso.

1.8 Il linguaggio MAO

Per lo studio dei fondamenti della programmazione utilizzeremo un linguaggio didattico appositamente progettato.

> **Definizione 1.12 – MAO – Modello Astratto Operazionale.** MAO è un linguaggio di programmazione imperativo semplificato, progettato a fini didattici. Pur essendo sintetico, MAO include tutti i costrutti fondamentali presenti nei linguaggi reali: dichiarazioni, assegnamenti, condizionali, cicli, funzioni e array.

La scelta di un linguaggio didattico permette di concentrarsi sui **concetti** della programmazione senza le complicazioni tecniche dei linguaggi industriali (gestione della memoria, librerie, ambienti di sviluppo, ecc.).

1.8.1 Costrutti principali

I costrutti fondamentali di MAO sono riassunti nella tabella seguente.

Costrutto	Sintassi	Descrizione
Dichiarazione	`int x = 5;`	Introduce una nuova variabile con tipo e valore iniziale
Assegnamento	`x := x + 1;`	Modifica il valore di una variabile già dichiarata
Condizionale	`if(x > 0){ ... } else { ... }`	Esegue un blocco in base a una condizione
Ciclo	`while(x > 0){ ... }`	Ripete un blocco finché la condizione è vera
Funzione	`int f(int a){ return a * 2; }`	Definisce un sottoprogramma con parametri e valore di ritorno
Array	`int[] A = int[5];`	Dichiara una sequenza indicizzata di valori

Tabella 2: Costrutti fondamentali del linguaggio MAO

> **Nota – Dichiarazione vs. Assegnamento.** In MAO la distinzione tra **dichiarazione** e **assegnamento** è resa esplicita dalla sintassi:
> - il simbolo `=` si usa esclusivamente nella **dichiarazione**, cioè quando si introduce una nuova variabile (ad esempio `int x = 5;`);
> - il simbolo `:=` si usa per l'**assegnamento**, cioè per modificare il valore di una variabile già esistente (ad esempio `x := x + 1;`).
>
> Questa convenzione, diversa da quella adottata dalla maggior parte dei linguaggi reali (che usano `=` per entrambe le operazioni), ha il vantaggio di rendere immediatamente visibile se un'istruzione sta creando una nuova variabile o modificandone una esistente.

Esempio 1.4 – Primo programma in MAO.

```
int x = 5;
int y = 3;
int somma = x + y;
```

Il programma dichiara due variabili intere `x` e `y`, poi dichiara una terza variabile `somma` il cui valore iniziale è la somma delle prime due. Al termine dell'esecuzione, `somma` contiene il valore 8.

Esempio 1.5 – Uso del ciclo while.

```
int n = 5;
int fatt = 1;
while(n > 1){
    fatt := fatt * n;
    n := n - 1;
}
```

Questo programma calcola il fattoriale di 5. Si noti l'uso di `:=` per gli assegnamenti all'interno del ciclo: le variabili `fatt` e `n` sono già state dichiarate e qui ne modifichiamo il valore.

1.9 Sintassi e Semantica

Lo studio di un linguaggio di programmazione si articola in due aspetti complementari ma distinti: la **sintassi**, che riguarda la forma delle frasi, e la **semantica**, che riguarda il loro significato.

Aspetto	Domanda chiave	Esempio
Sintassi	Come si scrive un programma valido?	`if(x > 0){ ... }`
Semantica	Cosa significa un programma valido?	Se $x > 0$, esegui il blocco

Tabella 3: I due livelli di analisi di un linguaggio

Definizione 1.13 – Sintassi. La sintassi di un linguaggio è l'insieme delle regole che stabiliscono quali sequenze di simboli costituiscono frasi (programmi) **ben formate**. In altri termini, la sintassi definisce la struttura grammaticale del linguaggio.

Definizione 1.14 – Semantica. La semantica di un linguaggio associa un **significato** a ciascuna frase sintatticamente corretta. Nel caso dei linguaggi di programmazione, la semantica specifica quale computazione viene eseguita da un programma: quali operazioni vengono svolte, in quale ordine e con quale effetto sullo stato della macchina.

La distinzione tra sintassi e semantica è cruciale: un programma può essere sintatticamente corretto ma semanticamente errato (ovvero, compila senza errori ma non fa ciò che vogliamo), oppure sintatticamente scorretto e quindi non eseguibile affatto.

Esempio 1.6 – Stesso significato, sintassi diversa. L'operazione «incrementa x di 1» si scrive in modo diverso a seconda del linguaggio:

Linguaggio	Sintassi
MAO	`x := x + 1;`
Python	`x = x + 1`
C/C++	`x++;`

Tabella 4: La stessa operazione in linguaggi diversi

Le tre istruzioni hanno la stessa **semantica** (incrementare il valore di x di un'unità) ma **sintassi** differente. Questo dimostra che la sintassi è una proprietà del linguaggio, mentre la semantica è legata al significato dell'operazione.

1.9.1 Perché una sintassi formale?

Nel linguaggio naturale, un interlocutore umano è in grado di interpretare frasi anche in presenza di errori:

Esempio 1.7 – Ambiguità del linguaggio naturale. La frase «Dmoani vado al mrae» è comprensibile per un essere umano come «Domani vado al mare», nonostante gli errori di ortografia. Un calcolatore, invece, non possiede questa capacità di interpretazione: richiede istruzioni scritte in modo **esatto** e **non ambiguo**.

Per questo motivo i linguaggi di programmazione hanno una sintassi **formale**, definita in modo rigoroso tramite **grammatiche formali**. Una grammatica formale specifica in modo preciso e completo quali sequenze di simboli appartengono al linguaggio e quali no.

Esempio 1.8 – Errori sintattici in MAO.

```
int x = ;        // ERRORE: manca l'espressione dopo =
if x > 0 { }    // ERRORE: mancano le parentesi tonde
int 3y = 10;     // ERRORE: identificatore non valido
```

Ciascuna di queste righe viola una regola sintattica di MAO e viene rifiutata prima ancora che il programma possa essere eseguito. Il compilatore (o l'interprete) segnala l'errore indicando quale regola è stata violata.

Nota. Le grammatiche formali non sono una peculiarità dei linguaggi di programmazione: anche le lingue naturali possiedono una grammatica (soggetto + verbo + complemento in italiano). La differenza fondamentale è che le grammatiche dei linguaggi naturali ammettono eccezioni e ambiguità, mentre quelle dei linguaggi formali sono **complete** e **non ambigue** per costruzione.

1.10 Obiettivi della programmazione

Scrivere un programma corretto non è sufficiente: un buon programma deve soddisfare diversi criteri di qualità.

- **Correttezza**: il programma deve risolvere il problema per cui è stato progettato, producendo l'output atteso per ogni input valido. La correttezza può essere verificata tramite **testing** (esecuzione su casi di prova) o dimostrata formalmente tramite **invarianti** e **precondizioni/postcondizioni**.

- **Efficienza**: il programma deve utilizzare le risorse computazionali (tempo di esecuzione e spazio in memoria) in modo ragionevole. Lo studio dell'efficienza è l'oggetto della **complessità computazionale**, trattata nella parte algoritmica del corso.
- **Leggibilità**: il codice deve essere comprensibile da altri programmatori (e dal programmatore stesso a distanza di tempo). Nomi di variabili significativi, commenti appropriati e una struttura logica chiara contribuiscono alla leggibilità.
- **Manutenibilità**: il programma deve essere facilmente modificabile e aggiornabile. Un codice ben strutturato e modulare facilita la correzione di errori e l'aggiunta di nuove funzionalità.

> **Nota.** Questi obiettivi possono talvolta entrare in conflitto: ad esempio, un'ottimizzazione aggressiva per l'efficienza può rendere il codice meno leggibile. La buona pratica di programmazione consiste nel trovare il giusto equilibrio tra questi criteri.

1.10.1 Verso la formalizzazione

Per poter studiare i linguaggi di programmazione con rigore matematico, è necessario formalizzarne sia la sintassi sia la semantica. Nel seguito del corso affronteremo questi temi in modo sistematico:

1. **Linguaggi formali e grammatiche**: definiremo gli strumenti matematici (alfabeti, stringhe, grammatiche) per descrivere la sintassi di un linguaggio.
2. **Semantica operazionale**: formalizzeremo il significato dei programmi tramite regole di inferenza che descrivono come un programma modifica lo stato della macchina durante l'esecuzione.
3. **Sistemi di tipi**: studieremo come verificare staticamente (cioè prima dell'esecuzione) che un programma rispetti vincoli di coerenza sui tipi dei dati.

Tutti questi aspetti saranno illustrati sul linguaggio MAO, che verrà introdotto gradualmente: prima nella sua versione base (MiniMao, con sole espressioni e comandi semplici), poi nella versione completa (con array, funzioni e ricorsione).

2 Linguaggi Formali e Grammatiche

2.1 Alfabeti, Stringhe e Linguaggi

2.1.1 Alfabeto

Definizione 2.1 – Alfabeto. Un **alfabeto** A è un insieme finito e non vuoto di simboli. I singoli elementi dell'alfabeto sono chiamati **simboli** o **caratteri**.

Esempio 2.1 – Alfabeti comuni.
- Alfabeto latino minuscolo: $A_1 = \{a, b, c, d, ..., z\}$
- Alfabeto binario: $A_2 = \{0, 1\}$
- Alfabeto di un linguaggio di programmazione: $A_3 = \{a, ..., z, 0, ..., 9, +, -, \times, ...\}$

2.1.2 Stringa

Definizione 2.2 – Stringa. Una **stringa** (o **parola**) su un alfabeto A è una sequenza finita di simboli appartenenti ad A. Formalmente, dati $a_1, a_2, ..., a_n \in A$ con $n \geq 0$, la sequenza $a_1 a_2 \cdots a_n$ è una stringa su A.

2.1.2.1 Lunghezza di una stringa

Definizione 2.3 – Lunghezza. La **lunghezza** di una stringa $s = a_1 a_2 \cdots a_n$ è il numero naturale n di simboli che la compongono e si denota con $|s|$.

Definizione 2.4 – Stringa vuota. Se $n = 0$ la stringa è detta **stringa vuota** e si denota con ε. Si ha $|\varepsilon| = 0$.

Esempio 2.2 – Lunghezze di stringhe. Sull'alfabeto $A = \{a, b, f, z\}$:
$|abfbz| = 5$
$|aaa| = 3$
$|\varepsilon| = 0$

2.1.3 Operazioni sulle stringhe

2.1.3.1 Concatenazione

Definizione 2.5 – Concatenazione. Date due stringhe $s = a_1 \cdots a_n$ e $t = b_1 \cdots b_m$ su un alfabeto A, la **concatenazione** di s e t è la stringa $s \cdot t = a_1 \cdots a_n b_1 \cdots b_m$. Si ha $|s \cdot t| = |s| + |t|$.

La stringa vuota ε è l'**elemento neutro** della concatenazione: per ogni stringa s, $s \cdot \varepsilon = \varepsilon \cdot s = s$.

Esempio 2.3 – Concatenazione. Sull'alfabeto $A = \{a, b, c\}$:
$ab \cdot ca = abca$
$ab \cdot \varepsilon = ab$

2.1.4 Insiemi di stringhe di lunghezza fissata

Definizione 2.6 – Insieme A^n. Dato un alfabeto A e un numero naturale $n \geq 0$, definiamo A^n come l'insieme di tutte e sole le stringhe su A che hanno lunghezza esattamente n.

Esempio 2.4 – Stringhe su $A = \{0, 1\}$. $A^0 = \{\varepsilon\}$
$A^1 = \{0, 1\}$
$A^2 = \{00, 01, 10, 11\}$
$A^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

In generale, se $|A| = k$, allora $|A^n| = k^n$.

2.1.5 Chiusura di Kleene

Definizione 2.7 – Chiusura di Kleene A^*. Dato un alfabeto A, la **chiusura di Kleene** (o **chiusura riflessiva e transitiva**) A^* è l'insieme di tutte le stringhe su A, inclusa la stringa vuota:

$$A^* = \bigcup_{n \geq 0} A^n = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots = \{\varepsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots$$

Nota – Cardinalità di A^*. Se A non è vuoto, A^* è un insieme **infinito numerabile**: contiene infinite stringhe, ma è possibile elencarle tutte (ad esempio ordinandole per lunghezza crescente e, a parità di lunghezza, in ordine lessicografico).

Definizione 2.8 – Chiusura positiva A^+. La **chiusura positiva** A^+ è l'insieme di tutte le stringhe non vuote su A:

$$A^+ = \bigcup_{n \geq 1} A^n = A^* \setminus \{\varepsilon\}$$

2.2 Linguaggi formali

Definizione 2.9 – Linguaggio formale. Un **linguaggio formale** L su un alfabeto A è un qualunque sottoinsieme di A^*:

$$L \subseteq A^*$$

Nota – Linguaggi particolari.
- Il **linguaggio vuoto** $\emptyset$ non contiene alcuna stringa (attenzione: $\emptyset \neq \{\varepsilon\}$).
- Il **linguaggio universale** A^* contiene tutte le possibili stringhe su A.
- Il linguaggio $\{\varepsilon\}$ contiene solo la stringa vuota.

Esempio 2.5 – Linguaggi sull'alfabeto binario. Sull'alfabeto $A = \{0, 1\}$:
- $L_1 = \{0, 1, 00, 11\}$ è un linguaggio finito.
- $L_2 = \{0^n 1^n \mid n \geq 1\} = \{01, 0011, 000111, ...\}$ è un linguaggio infinito. Contiene le stringhe formate da n zeri seguiti da n uni, con $n \geq 1$.

2.2.1 Descrizione dei linguaggi

Descrivere un linguaggio di programmazione significa specificare l'insieme delle stringhe ben formate (i **programmi**) del linguaggio. Poiché i programmi ammissibili sono tipicamente infiniti, non è possibile elencarli uno per uno: servono meccanismi finiti per descrivere insiemi infiniti.

Esistono due approcci fondamentali:

- **Metodo generativo**: si definisce una **grammatica** che genera tutte e sole le stringhe del linguaggio, partendo da un simbolo iniziale e applicando regole di riscrittura.
- **Metodo riconoscitivo**: si definisce un **automa** che, data una stringa, decide se essa appartiene o meno al linguaggio.

In questo capitolo ci concentriamo sul metodo generativo.

2.3 Grammatiche formali

Definizione 2.10 – Grammatica formale. Una **grammatica formale** è una tripla $G = (T, N, P)$ dove:

- T è un insieme finito di simboli **terminali** (l'alfabeto del linguaggio).
- N è un insieme finito di simboli **non terminali** (anche detti **categorie sintattiche** o **variabili**), con $T \cap N = \emptyset$.
- P è un insieme finito di **produzioni** (o **regole di riscrittura**).

Posto $S = T \cup N$ l'insieme di tutti i simboli, ogni produzione ha la forma:

$$\alpha ::= \beta \quad \text{con } \alpha \in S^* N S^* \text{ e } \beta \in S^*$$

dove α è detta **parte sinistra** e β è detta **parte destra**. La condizione $\alpha \in S^* N S^*$ impone che la parte sinistra contenga **almeno un non terminale**.

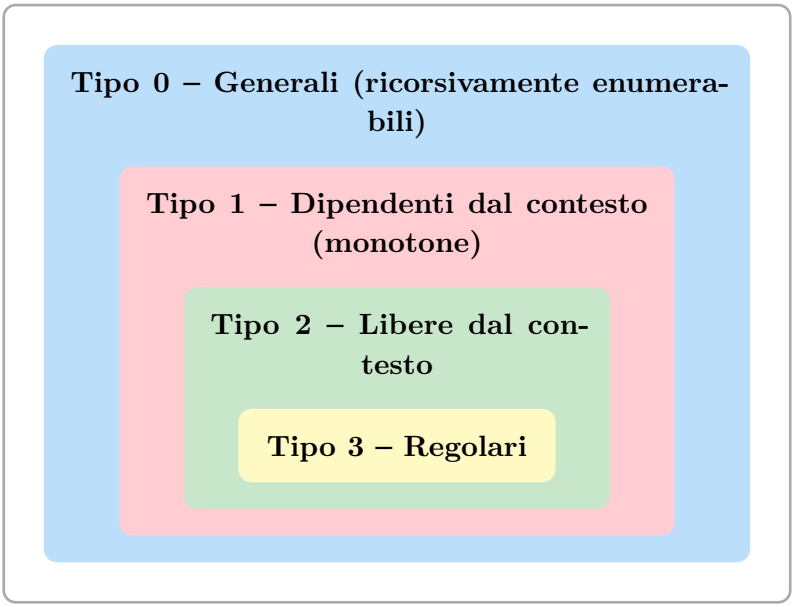
Nota – Simbolo iniziale. In molte formulazioni la grammatica è definita come una quadrupla $G = (T, N, S_0, P)$, dove $S_0 \in N$ è un **simbolo iniziale** (o **assioma**) distinto. Il linguaggio generato dalla grammatica è il linguaggio del simbolo S_0. Nella formulazione a tripla, il simbolo iniziale è indicato implicitamente come il primo non terminale delle produzioni.

Esempio 2.6 – Grammatica per stringhe di parentesi bilanciate. $T = \{(,)\}$, $N = \{\text{Par}\}$, con produzioni:
Par ::= ε
Par ::= (Par)
Par ::= Par Par

Questa grammatica genera tutte le stringhe di parentesi bilanciate: ε, ( ), (()), ( )( ), (()()), …

2.4 Gerarchia di Chomsky

Le grammatiche formali si classificano in base alla forma delle loro produzioni. La **gerarchia di Chomsky** definisce quattro classi, ognuna contenuta nella precedente.



Definizione 2.11 – Tipo 0 – Grammatiche generali. Le produzioni hanno la forma $\alpha ::= \beta$ con $\alpha \in S^*NS^*$ e $\beta \in S^*$, senza alcuna restrizione ulteriore. I linguaggi generati sono detti **ricorsivamente enumerabili** e sono riconosciuti dalle macchine di Turing.

Definizione 2.12 – Tipo 1 – Grammatiche dipendenti dal contesto. Le produzioni hanno la forma $\alpha ::= \beta$ con il vincolo $|\alpha| \leq |\beta|$ (le produzioni non possono accorciare la stringa). Si ammette l'eccezione $S_0 ::= \varepsilon$ solo se S_0 non compare nella parte destra di alcuna produzione.

I linguaggi generati sono detti **dipendenti dal contesto** e sono riconosciuti dagli automi lineari limitati.

Definizione 2.13 – Tipo 2 – Grammatiche libere dal contesto (context-free). Le produzioni hanno la forma $X ::= \beta$ dove $X \in N$ è un singolo non terminale e $\beta \in S^*$. La riscrittura di X non dipende dal contesto in cui X appare.

I linguaggi generati sono detti **liberi dal contesto** (context-free) e sono riconosciuti dagli automi a pila.

Nota – Importanza delle grammatiche libere dal contesto. La maggior parte dei linguaggi di programmazione ha una **sintassi** descrivibile con grammatiche libere dal contesto (tipo 2). Questo è il tipo di grammatica su cui ci concentreremo nel corso.

Definizione 2.14 – Tipo 3 – Grammatiche regolari. Le produzioni hanno una delle seguenti forme:

- $X ::= aY$ oppure $X ::= a$ (**regolari a destra**), oppure
- $X ::= Ya$ oppure $X ::= a$ (**regolari a sinistra**)

dove $X, Y \in N$ e $a \in T$. I linguaggi generati sono detti **regolari** e sono riconosciuti dagli automi a stati finiti. Possono anche essere descritti da **espressioni regolari**.

2.5 Backus-Naur Form (BNF)

La **Backus-Naur Form** (BNF) è una notazione compatta per scrivere grammatiche libere dal contesto. Le convenzioni sono:

- I non terminali sono racchiusi tra parentesi angolari $\langle \cdots \rangle$ oppure scritti con iniziale maiuscola.
- Il simbolo ::= separa la parte sinistra dalla parte destra.
- Il simbolo | separa le alternative: tutte le produzioni con lo stesso non terminale a sinistra vengono raggruppate in una sola riga.

> **Esempio 2.7 – Grammatica per indicazioni stradali in BNF.** ⟨Direzione⟩ ::= sinistra | destra
> ⟨Consiglio⟩ ::= svolta a ⟨Direzione⟩ | prosegui dritto
> ⟨Percorso⟩ ::= ⟨Consiglio⟩ | ⟨Consiglio⟩, poi ⟨Percorso⟩
>
> Questa grammatica genera frasi come: «svolta a sinistra, poi prosegui dritto».

2.5.1 Espressioni aritmetiche in BNF

> **Esempio 2.8 – Grammatica per le espressioni aritmetiche.** Dato l'alfabeto dei terminali $T = \{0, 1, 2, ..., 9, +, \times\}$, definiamo la grammatica $G = (T, N, P)$ con $N = \{\text{Cifra}, \text{Num}, \text{Op}, \text{Esp}\}$ e le produzioni:
>
> Cifra ::= 0 | 1 | 2 | … | 9
> Num ::= Cifra | Num Cifra
> Op ::= + | ×
> Esp ::= Num | Esp Op Esp
>
> Ogni produzione definisce come costruire espressioni valide:
> - **Cifra** genera una singola cifra decimale.
> - **Num** genera un numero naturale come sequenza di cifre (definizione ricorsiva).
> - **Op** genera un operatore aritmetico.
> - **Esp** genera un'espressione: un numero oppure due espressioni collegate da un operatore.

> **Nota.** Il non terminale Esp è la categoria sintattica principale: il linguaggio delle espressioni aritmetiche è $L(\text{Esp})$. Vedremo nel prossimo paragrafo come formalizzare il concetto di «linguaggio generato da un non terminale».

2.6 Derivazioni e Linguaggi Generati

Il meccanismo fondamentale delle grammatiche formali è la **derivazione**: si parte da un non terminale e, applicando ripetutamente le produzioni, si ottengono stringhe di terminali. In questa sezione formalizziamo questo processo.

2.6.1 Derivazione immediata

> **Definizione 2.15 – Derivazione immediata.** Data una grammatica $G = (T, N, P)$ con $S = T \cup N$, e date due stringhe $\beta, \beta' \in S^*$, si dice che β' è un **derivato immediato** di β, e si scrive:
>
> $$\beta \to \beta'$$
>
> se e solo se esistono stringhe $\beta_1, \beta_2 \in S^*$, un non terminale $X \in N$ e una produzione $X ::= \alpha \in P$ tali che:

$$\beta = \beta_1 X \beta_2 \quad \text{e} \quad \beta' = \beta_1 \alpha \beta_2$$

In altre parole, β' si ottiene da β sostituendo un'occorrenza del non terminale X con la parte destra α di una sua produzione, lasciando invariato il contesto β_1 e β_2.

Esempio 2.9 – Derivazione immediata. Data la grammatica con le produzioni:
Esp ::= Num | Esp Op Esp
Op ::= + | ×
Num ::= 0 | 1 | … | 9

Partiamo dalla stringa β = Esp + Esp Op 2. Applicando la produzione Esp ::= Num all'occorrenza più a sinistra di Esp, otteniamo:

$$\text{Esp} + \text{Esp Op 2} \to \text{Num} + \text{Esp Op 2}$$

Qui $\beta_1 = \varepsilon$, X = Esp, α = Num, β_2 = + Esp Op 2.

Nota. Se una stringa contiene più non terminali, ad ogni passo possiamo scegliere **quale** non terminale espandere e **quale** produzione applicare. Questa libertà di scelta porta, in generale, a derivazioni diverse per la stessa stringa.

2.6.2 Derivazione (in zero o più passi)

Definizione 2.16 – Derivazione. Data una grammatica $G = (T, N, P)$, e date due stringhe $\beta, \beta' \in S^*$, si dice che β' è un **derivato** di β, e si scrive:

$$\beta \xrightarrow{*} \beta'$$

se e solo se esiste una sequenza finita (eventualmente vuota) di derivazioni immediate:

$$\beta = \beta_0 \to \beta_1 \to \beta_2 \to \cdots \to \beta_n = \beta' \quad (n \geq 0)$$

Il numero n è detto **lunghezza della derivazione**.

- Se $n = 0$, allora $\beta = \beta'$ (ogni stringa deriva da se stessa: **riflessività**).
- Se $\beta \xrightarrow{*} \gamma$ e $\gamma \xrightarrow{*} \beta'$, allora $\beta \xrightarrow{*} \beta'$ (**transitività**).

Esempio 2.10 – Derivazione passo per passo. Con la grammatica delle espressioni, deriviamo la stringa $3 + 5$:

$$\begin{aligned} & \text{Esp} \\ \to\ & \text{Esp Op Esp} \\ \to\ & \text{Num Op Esp} \\ \to\ & \text{3 Op Esp} \\ \to\ & 3 + \text{Esp} \\ \to\ & 3 + \text{Num} \\ \to\ & 3 + 5 \end{aligned}$$

Poiché Esp $\xrightarrow{*}$ $3 + 5$ e $3 + 5 \in T^*$, la stringa $3 + 5$ appartiene al linguaggio generato da Esp.

2.7 Linguaggio generato da un non terminale

Definizione 2.17 – Linguaggio generato. Data una grammatica $G = (T, N, P)$ e un non terminale $X \in N$, il **linguaggio generato** da X, scritto $L(X)$, è l'insieme di tutte e sole le stringhe di terminali derivabili da X:

$$L(X) = \left\{ w \in T^* \mid X \xrightarrow{*} w \right\}$$

Quando la grammatica ha un simbolo iniziale S_0, il **linguaggio generato dalla grammatica** è $L(G) = L(S_0)$.

Nota. Si noti che $L(X) \subseteq T^*$: il linguaggio contiene solo stringhe composte esclusivamente da simboli terminali. Le stringhe intermedie che contengono ancora non terminali (dette **forme sentenziali**) non fanno parte del linguaggio.

2.7.1 Appartenenza al linguaggio

Data una grammatica $G = (T, N, P)$ e una stringa $w \in T^*$, per stabilire se $w \in L(X)$ si ragiona nel modo seguente:

- $w \in L(X)$ se e solo se partendo dal simbolo X è possibile, applicando una o più produzioni, ottenere la stringa w composta da soli terminali.
- $w \notin L(X)$ se e solo se **qualunque** sequenza di applicazioni di produzioni a partire da X non è in grado di generare w.

Osservazione. Dimostrare che $w \in L(X)$ è relativamente semplice: basta esibire una derivazione $X \xrightarrow{*} w$. Dimostrare che $w \notin L(X)$ è più difficile, perché richiede di escludere **tutte** le possibili derivazioni.

2.7.2 Esempio completo: numeri naturali senza zeri iniziali

Esempio 2.11 – Grammatica per numeri naturali senza zeri iniziali. Consideriamo la grammatica dell'esempio precedente per le espressioni aritmetiche. La produzione Num ::= Cifra | Num Cifra permette di generare stringhe come 007, che non è una rappresentazione canonica di un numero naturale.

Per ottenere una grammatica che non generi zeri iniziali, modifichiamo le produzioni:

NonZero ::= 1 | 2 | 3 | … | 9
Cifra ::= 0 | NonZero
Pos ::= NonZero | Pos Cifra
Num ::= 0 | Pos

In questa grammatica:

- **NonZero** genera una cifra diversa da zero.
- **Cifra** genera una cifra qualsiasi (0–9).
- **Pos** genera un numero positivo: la prima cifra è necessariamente diversa da zero (NonZero), mentre le cifre successive possono essere qualsiasi (Cifra).
- **Num** genera lo zero oppure un numero positivo.

Verifichiamo: L(Num) contiene 0, 5, 42, 100, ma **non** contiene 007 né 00.

2.8 Derivazioni canoniche

Abbiamo visto che, quando una forma sentenziale contiene più non terminali, si può scegliere quale espandere per primo. Le **derivazioni canoniche** eliminano questa libertà di scelta fissando una strategia deterministica.

2.8.1 Derivazione canonica sinistra

Definizione 2.18 – Derivazione canonica sinistra (leftmost derivation). Una derivazione $\beta_0 \to \beta_1 \to \cdots \to \beta_n$ è **canonica sinistra** se, ad ogni passo $\beta_i \to \beta_{i+1}$, il non terminale sostituito è sempre quello **più a sinistra** nella stringa β_i.

2.8.2 Derivazione canonica destra

Definizione 2.19 – Derivazione canonica destra (rightmost derivation). Una derivazione $\beta_0 \to \beta_1 \to \cdots \to \beta_n$ è **canonica destra** se, ad ogni passo $\beta_i \to \beta_{i+1}$, il non terminale sostituito è sempre quello **più a destra** nella stringa β_i.

Esempio 2.12 – Derivazioni canoniche a confronto. Data la grammatica:
Esp ::= Num | Esp + Esp
Num ::= 0 | 1 | 2 | … | 9

Deriviamo la stringa $3 + 5 + 2$.

Derivazione canonica sinistra (ad ogni passo si espande il non terminale più a sinistra):

$$\begin{aligned} & \text{Esp} \\ \to\ & \text{Esp} + \text{Esp} \\ \to\ & \text{Esp} + \text{Esp} + \text{Esp} \\ \to\ & \text{Num} + \text{Esp} + \text{Esp} \\ \to\ & 3 + \text{Esp} + \text{Esp} \\ \to\ & 3 + \text{Num} + \text{Esp} \\ \to\ & 3 + 5 + \text{Esp} \\ \to\ & 3 + 5 + \text{Num} \\ \to\ & 3 + 5 + 2 \end{aligned}$$

Derivazione canonica destra (ad ogni passo si espande il non terminale più a destra):

$$\begin{aligned} & \text{Esp} \\ \to\ & \text{Esp} + \text{Esp} \\ \to\ & \text{Esp} + \text{Num} \\ \to\ & \text{Esp} + 2 \\ \to\ & \text{Esp} + \text{Esp} + 2 \\ \to\ & \text{Esp} + \text{Num} + 2 \\ \to\ & \text{Esp} + 5 + 2 \\ \to\ & \text{Num} + 5 + 2 \\ \to\ & 3 + 5 + 2 \end{aligned}$$

Nota. Entrambe le derivazioni producono la stessa stringa $3 + 5 + 2$, ma con ordini di sostituzione diversi. La derivazione canonica sinistra e quella destra rappresentano due strategie sistematiche per enumerare le derivazioni.

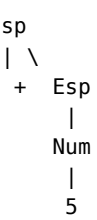
2.9 Alberi di derivazione

Derivazioni diverse (per esempio la canonica sinistra e la canonica destra) possono corrispondere alla stessa «struttura» della derivazione. L'**albero di derivazione** (o **parse tree**) cattura esattamente questa struttura, astraendo dall'ordine in cui le produzioni vengono applicate.

Definizione 2.20 – Albero di derivazione (parse tree). Data una grammatica libera dal contesto $G = (T, N, P)$ e un non terminale $X \in N$, un **albero di derivazione** per una stringa $w \in L(X)$ è un albero ordinato con le seguenti proprietà:

1. La **radice** è etichettata con X.
2. Ogni **nodo interno** è etichettato con un non terminale $Y \in N$.
3. Le **foglie** sono etichettate con simboli terminali $a \in T$ oppure con ε.
4. Se un nodo interno è etichettato con Y e i suoi figli (da sinistra a destra) sono etichettati con $X_1, X_2, ..., X_k$, allora $Y ::= X_1 X_2 \cdots X_k$ è una produzione di P.
5. La stringa ottenuta leggendo le foglie da sinistra a destra è w (detta **frontiera** dell'albero).

Esempio 2.13 – Albero di derivazione per la stringa 3 + 5. Con la grammatica Esp ::= Num | Esp + Esp e Num ::= 0 | … | 9, l'albero per $3 + 5$ è:



Lettura dell'albero:

- La radice Esp si espande con la produzione Esp ::= Esp + Esp.
- Il figlio sinistro Esp si espande con Esp ::= Num, poi Num ::= 3.
- Il figlio destro Esp si espande con Esp ::= Num, poi Num ::= 5.
- La frontiera (foglie da sinistra a destra) è $3 + 5$.

Osservazione. Derivazioni canoniche diverse (sinistra e destra) possono produrre lo **stesso albero** di derivazione. L'albero cattura la **struttura** della derivazione, non l'**ordine** delle sostituzioni. In particolare, per una grammatica libera dal contesto, esiste una corrispondenza biunivoca tra alberi di derivazione e derivazioni canoniche sinistre (e analogamente con le derivazioni canoniche destre).

2.9.1 Valutazione basata sull'albero

Quando l'albero di derivazione rappresenta un'espressione, la sua struttura determina l'ordine di valutazione:

- Si valutano prima i **sottoalberi** (ricorsivamente, dalle foglie verso la radice).
- Poi si applica l'operatore del nodo corrente ai risultati dei sottoalberi.

Esempio 2.14 – Valutazione dell'espressione 3 + 5. Dall'albero dell'esempio precedente:
1. Valuta il sottoalbero sinistro: Num → 3, valore = 3.
2. Valuta il sottoalbero destro: Num → 5, valore = 5.
3. Applica l'operatore +: $3 + 5 = 8$.

2.10 Ambiguità

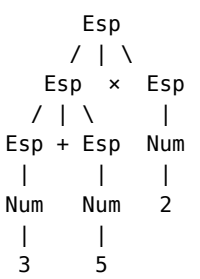
Definizione 2.21 – Grammatica ambigua. Una grammatica G è **ambigua** se esiste almeno una stringa $w \in L(G)$ che ammette **due o più alberi di derivazione distinti**. Equivalentemente, G è ambigua se esiste una stringa che ammette due derivazioni canoniche sinistre distinte (o due derivazioni canoniche destre distinte).

L'ambiguità è un problema grave perché alberi di derivazione diversi possono assegnare **significati diversi** alla stessa stringa.

Esempio 2.15 – Ambiguità nella stringa $3 + 5 \times 2$. Consideriamo la grammatica:
Esp ::= Num | Esp + Esp | Esp × Esp
Num ::= 0 | 1 | … | 9

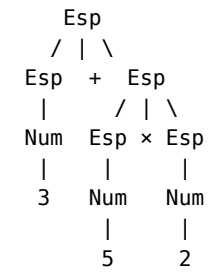
La stringa $3 + 5 \times 2$ ammette due alberi di derivazione distinti.

Albero 1 – interpreta come $(3 + 5) \times 2$:



Valore: $(3 + 5) \times 2 = 16$

Albero 2 – interpreta come $3 + (5 \times 2)$:



Valore: $3 + (5 \times 2) = 13$

La stessa stringa ha due valutazioni diverse: 16 oppure 13, a seconda dell'albero scelto. Per un linguaggio di programmazione, questa situazione è inaccettabile.

2.11 Risoluzione dell'ambiguità

Per eliminare l'ambiguità da una grammatica si possono adottare diverse strategie.

2.11.1 Introduzione di livelli di precedenza

Si ristruttura la grammatica introducendo non terminali aggiuntivi che codificano la **precedenza** e l'**associatività** degli operatori. L'idea è che gli operatori a precedenza più alta vengano «catturati» più in profondità nell'albero.

> **Esempio 2.16 – Grammatica non ambigua con precedenza.** Esp ::= Term | Esp + Term
> Term ::= Factor | Term × Factor
> Factor ::= Num | (Esp)
> Num ::= 0 | 1 | … | 9
>
> In questa grammatica:
> - **Esp** gestisce l'addizione: un'espressione è un termine, oppure un'espressione seguita da + e un termine. L'addizione è **associativa a sinistra**.
> - **Term** gestisce la moltiplicazione: un termine è un fattore, oppure un termine seguito da × e un fattore. La moltiplicazione è **associativa a sinistra** e ha **precedenza maggiore** rispetto all'addizione.
> - **Factor** gestisce le «unità atomiche»: un numero oppure un'espressione racchiusa tra parentesi.
>
> Con questa grammatica, la stringa $3 + 5 \times 2$ ha un **unico** albero di derivazione, che corrisponde all'interpretazione $3 + (5 \times 2) = 13$, rispettando la precedenza usuale della moltiplicazione sull'addizione.

2.11.2 Uso delle parentesi

Un altro metodo per risolvere l'ambiguità consiste nell'introdurre le **parentesi** nella grammatica per forzare esplicitamente l'ordine di valutazione.

> **Esempio 2.17 – Grammatica con parentesi obbligatorie.** Esp ::= Num | (Esp Op Esp)
> Op ::= + | ×
> Num ::= 0 | 1 | … | 9
>
> Con questa grammatica, ogni operazione binaria deve essere racchiusa tra parentesi, quindi non c'è ambiguità: $((3 + 5) \times 2)$ e $(3 + (5 \times 2))$ sono stringhe diverse.

> **Nota – Linguaggi inerentemente ambigui.** Non sempre è possibile eliminare l'ambiguità modificando la grammatica. Esistono **linguaggi inerentemente ambigui**: linguaggi per i quali **ogni** grammatica che li genera è ambigua. Un esempio classico è il linguaggio $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$. Tuttavia, per i linguaggi di programmazione questo problema tipicamente non si pone.

3 Semantica Operazionale

Le grammatiche formali, introdotte nel capitolo precedente, possono essere lette in due modi complementari. Questa doppia lettura è alla base della semantica formale dei linguaggi di programmazione.

> **Definizione 3.1 – Lettura generativa e lettura induttiva.** Data una produzione come $S ::= ab \mid aSb$, possiamo interpretarla in due modi:
> - **Lettura generativa (produttiva)**: la grammatica è vista come un insieme di regole di riscrittura. La produzione si legge da sinistra verso destra: ogni volta che incontro il simbolo S, posso rimpiazzarlo con ab oppure con aSb. Questa lettura descrive come *generare* stringhe.
> - **Lettura induttiva (costruttiva)**: la grammatica è una definizione induttiva. La produzione si legge da destra verso sinistra: la clausola $S ::= ab$ afferma che $ab \in L(S)$ (caso base), mentre $S ::= aSb$ afferma che se $w \in L(S)$ allora anche $awb \in L(S)$ (passo induttivo). Questa lettura descrive come *costruire* l'insieme delle stringhe valide.

La lettura induttiva è particolarmente importante perché ci permette di utilizzare le **regole di inferenza** per definire la semantica di un linguaggio di programmazione in modo rigoroso.

3.1 Regole di inferenza

Per definire la semantica del linguaggio MAO utilizzeremo le regole di inferenza, uno strumento formale che permette di derivare nuovi giudizi (conclusioni) a partire da giudizi già noti (premesse).

> **Definizione 3.2 – Regola di inferenza.** Una regola di inferenza ha la seguente forma: date premesse $p_1, ..., p_n$ e una conclusione q, scriviamo:
>
> $$\frac{p_1 \ \cdots \ p_n}{q} \quad \text{(Nome-Regola)}$$
>
> La regola si legge: «se tutte le premesse $p_1, ..., p_n$ sono vere, allora si può trarre la conclusione q».

Quando le premesse sono vuote, la regola si chiama assioma: una verità che vale senza bisogno di giustificazione.

> **Definizione 3.3 – Assioma.** Un assioma è una regola di inferenza senza premesse:
>
> $$\frac{}{q} \quad \text{(Nome-Assioma)}$$
>
> L'assioma afferma che q è vero incondizionatamente.

3.1.1 Produzioni come regole di inferenza

Le produzioni di una grammatica possono essere viste come regole di inferenza. Consideriamo una grammatica per le espressioni aritmetiche con le produzioni:

$\text{Exp} ::= \text{Exp} + \text{Pro} \mid \text{Pro} \qquad \text{Pro} ::= \text{Pro} \times \text{Cifra} \mid \text{Cifra} \qquad \text{Cifra} ::= 0 \mid 1 \mid ... \mid 9$

Ogni produzione corrisponde a una regola di inferenza letta induttivamente:

$$\dfrac{y \in L(\text{Exp}) \qquad z \in L(\text{Pro})}{y + z \in L(\text{Exp})}\ (\text{Exp-Sum}) \qquad \dfrac{z \in L(\text{Pro})}{z \in L(\text{Exp})}\ (\text{Exp-Pro})$$

$$\dfrac{z \in L(\text{Pro}) \qquad c \in L(\text{Cifra})}{z \times c \in L(\text{Pro})}\ (\text{Pro-Mul}) \qquad \dfrac{}{5 \in L(\text{Cifra})}\ (\text{Cifra-5})$$

Ad esempio, la regola (Exp-Sum) si legge: «se y è un'espressione e z è un prodotto, allora $y + z$ è un'espressione». La regola (Cifra-5) è un assioma: 5 è una cifra senza bisogno di ulteriori giustificazioni.

3.2 Sistemi logici

Definizione 3.4 – Sistema logico. Un **sistema logico** è un insieme di regole di inferenza (e assiomi) che possono essere applicate per dimostrare la validità di formule, dette *giudizi*. Fissato un sistema logico, diciamo che un giudizio q è **derivabile** se esiste una sequenza di applicazioni di regole che lo dimostra.

Esempio 3.1 – Derivazione in un sistema logico. Consideriamo il sistema logico associato alla grammatica delle espressioni aritmetiche mostrata sopra. Vogliamo mostrare che 2×3 è un'espressione valida, cioè che $2 \times 3 \in L(\text{Exp})$.

Per farlo, costruiamo una derivazione applicando le regole dal basso verso l'alto:
1. $2 \in L(\text{Cifra})$ per l'assioma (Cifra-2), e quindi $2 \in L(\text{Pro})$ per (Pro-Cifra)
2. $3 \in L(\text{Cifra})$ per l'assioma (Cifra-3)
3. Da $2 \in L(\text{Pro})$ e $3 \in L(\text{Cifra})$, otteniamo $2 \times 3 \in L(\text{Pro})$ per la regola (Pro-Mul)
4. Da $2 \times 3 \in L(\text{Pro})$, otteniamo $2 \times 3 \in L(\text{Exp})$ per la regola (Exp-Pro)

3.2.1 Derivazione

Definizione 3.5 – Derivazione. Una **derivazione** nel sistema logico è una sequenza di passaggi che, partendo dagli assiomi e applicando le regole di inferenza, giustifica una certa conclusione. Si scrive $d \vdash q$ e si legge «la derivazione d dimostra il giudizio q», oppure «d è una derivazione per q».

Le derivazioni sono definite ricorsivamente:
- **Caso base**: ogni assioma del sistema logico è una derivazione per la sua conclusione q.
- **Passo induttivo**: se esiste una regola

$$\dfrac{p_1 \ \cdots \ p_n}{q}$$

del sistema logico, e le premesse sono derivabili ($d_1 \vdash p_1, ..., d_n \vdash p_n$), allora la composizione delle sotto-derivazioni è una derivazione per q.

Esempio 3.2 – Derivazione per $2 times 3 in L("Exp")$. Rappresentiamo la derivazione come albero di regole applicate (le foglie sono assiomi):

$$\dfrac{\dfrac{}{2 \in L(\text{Cifra})}}{2 \in L(\text{Pro})} \qquad \dfrac{}{3 \in L(\text{Cifra})}$$
$$2 \times 3 \in L(\text{Pro})$$

Da questa derivazione, applicando (Exp-Pro), concludiamo $2 \times 3 \in L(\text{Exp})$.

3.2.2 Alberi di derivazione

Le derivazioni formano naturalmente una **struttura ad albero**, anche se in questo caso la radice (la conclusione finale) è posta verso il basso, e le foglie (gli assiomi) sono in alto. Ogni nodo interno corrisponde all'applicazione di una regola, e i suoi figli sono le sotto-derivazioni delle premesse.

> **Nota.** Non confondere gli alberi di derivazione della semantica (che crescono verso il basso con la conclusione in fondo) con gli alberi di derivazione delle grammatiche (capitolo 2), che hanno la radice in alto.

3.2.3 Grammatiche come sistemi logici

Possiamo formalizzare il legame tra grammatiche e sistemi logici. Invece di usare rappresentazioni grafiche colorate, introduciamo le formule $x \in L(X)$, con il significato che «la stringa x appartiene al linguaggio generato dal non-terminale X».

Ad ogni produzione della grammatica della forma $X ::= \omega_0 X_1 \omega_1 X_2 \omega_2 ... X_n \omega_n$ (dove $\omega_i \in T^*$ sono stringhe di terminali e $X_i \in N$ sono non-terminali) associamo una regola di inferenza:

$$\frac{x_1 \in L(X_1) \quad x_2 \in L(X_2) \quad ... \quad x_n \in L(X_n)}{\omega_0 x_1 \omega_1 x_2 \omega_2 ... x_n \omega_n \in L(X)}$$

3.2.4 Valutazione di espressioni

I sistemi logici permettono non solo di stabilire se una stringa appartiene a un linguaggio, ma anche di assegnare un **significato** (valore) alle espressioni, seguendone la struttura grammaticale.

> **Esempio 3.3 – Grammatica ambigua delle espressioni.** Riprendiamo la grammatica ambigua delle espressioni con sole cifre:
>
> $E ::= 0 \mid 1 \mid 2 \mid ... \mid 9 \mid E + E \mid E \times E$
>
> Per ogni espressione w definiamo un predicato di valutazione $w \Downarrow v$, che si legge «il termine w ha valore v».

Definiamo le regole di valutazione come segue. Per le cifre abbiamo assiomi (una per ciascuna cifra da 0 a 9):

$$\frac{}{0 \Downarrow 0} \quad \text{(Val-0)}$$

$$\frac{}{1 \Downarrow 1}\ (\text{Val-1}) \qquad \ldots \qquad \frac{}{9 \Downarrow 9}\ (\text{Val-9})$$

Per le operazioni abbiamo regole con premesse:

$$\frac{x_1 \Downarrow n_1 \quad x_2 \Downarrow n_2 \quad n = n_1 + n_2}{x_1 + x_2 \Downarrow n}\ (\text{Val-Sum}) \qquad \frac{x_1 \Downarrow n_1 \quad x_2 \Downarrow n_2 \quad n = n_1 \times n_2}{x_1 \times x_2 \Downarrow n}\ (\text{Val-Mul})$$

Esempio 3.4 – Derivazione della valutazione di $1 + (3 \times 2)$. Vogliamo derivare $1 + (3 \times 2) \Downarrow 7$. Costruiamo l'albero di derivazione:

$$\dfrac{\dfrac{}{1 \Downarrow 1} \quad \dfrac{\dfrac{}{3 \Downarrow 3} \quad \dfrac{}{2 \Downarrow 2} \quad 6 = 3 \times 2}{3 \times 2 \Downarrow 6} \quad 7 = 1 + 6}{1 + (3 \times 2) \Downarrow 7}$$

La derivazione procede come segue:
1. Per l'assioma (Val-1): $1 \Downarrow 1$
2. Per l'assioma (Val-3): $3 \Downarrow 3$ e per (Val-2): $2 \Downarrow 2$
3. Per la regola (Val-Mul): poiché $3 \Downarrow 3$ e $2 \Downarrow 2$ e $6 = 3 \times 2$, concludiamo $3 \times 2 \Downarrow 6$
4. Per la regola (Val-Sum): poiché $1 \Downarrow 1$ e $3 \times 2 \Downarrow 6$ e $7 = 1 + 6$, concludiamo $1 + (3 \times 2) \Downarrow 7$

3.3 Induzione

Definizione 3.6. L'**induzione** è un principio fondamentale che permette di trattare insiemi infiniti di oggetti attraverso un numero finito di regole o casi.

Il principio di induzione ha tre aspetti fondamentali:
- **Costruzione**: permette di costruire un insieme infinito di oggetti mediante un numero finito di regole (ad esempio, l'insieme dei numeri naturali si definisce con due regole: 0 è un naturale, e se n è un naturale allora $n + 1$ è un naturale).
- **Definizione di funzioni**: permette di definire il comportamento di una funzione su un insieme infinito, descrivendo solo un numero finito di casi.
- **Dimostrazione**: permette di dimostrare che una proprietà vale per tutti gli elementi di un insieme infinito, esaminando un numero finito di casi.

3.3.1 Induzione matematica

L'induzione matematica permette di dimostrare che una proprietà $P(n)$ vale per tutti i numeri naturali $n \geq n_0$.

Definizione 3.7 – Principio di induzione matematica. Per dimostrare $P(n)$ per ogni $n \geq n_0$:
- **Caso base**: dimostrare che $P(n_0)$ vale.
- **Passo induttivo**: preso un generico $n \geq n_0$, assumendo che $P(n)$ sia vera (**ipotesi induttiva**), dimostrare che $P(n + 1)$ è vera.

Esempio 3.5 – Somma dei primi n naturali positivi. Dimostriamo che per ogni naturale positivo n:

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$$

Caso base ($n = 1$): $1 = \frac{1 \cdot 2}{2} = 1$. Verificato.

Passo induttivo: assumiamo che la formula valga per n (ipotesi induttiva):

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$$

Dimostriamo che vale per $n + 1$:

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

L'ultimo passaggio si ottiene raccogliendo $(n + 1)$ a fattor comune. La formula è quindi dimostrata per $n + 1$.

3.3.2 Induzione strutturale

L'induzione strutturale estende il principio di induzione matematica alle stringhe di un linguaggio generato da una grammatica. Invece di indurre sui numeri naturali, si induce sulla struttura delle stringhe.

Definizione 3.8 – Principio di induzione strutturale. Per dimostrare una proprietà $P(w)$ per tutte le stringhe $w \in L(S)$ generate da una grammatica:

- **Casi base**: dimostrare $P(\omega)$ per le stringhe $\omega \in T^*$ che compaiono nella parte destra delle produzioni *atomiche* (cioè produzioni della forma $X ::= \omega$ senza non-terminali a destra).
- **Casi induttivi**: per ogni produzione non atomica della forma $X ::= \omega_0 Y_1 \omega_1 ... Y_n \omega_n$, assumendo che $P(s_1), ..., P(s_n)$ valgano per le stringhe s_i generate dai non-terminali Y_i (ipotesi induttiva), dimostrare che $P(\omega_0 s_1 \omega_1 ... s_n \omega_n)$ vale.

3.3.3 Induzione sulle derivazioni

L'induzione sulle derivazioni permette di dimostrare proprietà valide per tutti i giudizi derivabili in un sistema logico. A differenza dell'induzione strutturale (che lavora sulla struttura delle stringhe), qui si lavora sulla **struttura delle derivazioni** stesse.

Definizione 3.9 – Principio di induzione sulle derivazioni. Sia $\mathcal{S}$ un sistema logico e P una proprietà sui giudizi. Per dimostrare che $P(J)$ vale per ogni giudizio J derivabile in $\mathcal{S}$:

Casi base (assiomi): Per ogni assioma

$$\frac{}{q}$$

del sistema, dimostrare che $P(q)$ vale.

Casi induttivi (regole): Per ogni regola

$$\frac{p_1 \ \cdots \ p_n}{q}$$

del sistema, assumendo che $P(p_1), ..., P(p_n)$ valgano (**ipotesi induttiva**), dimostrare che $P(q)$ vale.

Nota – Intuizione. L'induzione sulle derivazioni riflette il modo in cui i giudizi vengono costruiti: partendo dagli assiomi (verità di base) e applicando regole di inferenza. Se una proprietà vale per le «fondamenta» (assiomi) e si «propaga» attraverso le regole, allora vale per tutto ciò che è derivabile.

Esempio 3.6 – Applicazione: correttezza della valutazione. Consideriamo il sistema logico per la valutazione di espressioni aritmetiche con le regole:

$$\frac{}{n \Downarrow n} \quad \text{(Val-Num)}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2} \quad \text{(Val-Sum)}$$

Vogliamo dimostrare: «Se $e \Downarrow v$ è derivabile, allora v è un numero intero».

Caso base (Val-Num): $n \Downarrow n$. Il valore n è un numero intero per definizione.

Caso induttivo (Val-Sum): Assumiamo (ipotesi induttiva) che v_1 e v_2 siano interi. Allora $v_1 + v_2$ è la somma di due interi, quindi è un intero.

Esempio 3.7 – Confronto dei tre tipi di induzione.

Induzione matematica	**Induzione strutturale**	**Induzione sulle derivazioni**
Sui numeri naturali $\mathbb{N}$	Sulle stringhe di $L(G)$	Sui giudizi derivabili
Caso base: $n = 0$	Caso base: produzioni atomiche	Caso base: assiomi
Passo: $n \to n + 1$	Passo: produzioni ricorsive	Passo: regole di inferenza
Esempio: somma dei primi n	Esempio: $\#_{a(w)} = \#_{b(w)}$	Esempio: correttezza semantica

Questa tecnica sarà fondamentale per dimostrare proprietà della semantica di MAO, come ad esempio:

- **Determinismo**: se $\langle e, \rho, \sigma \rangle \Downarrow v_1$ e $\langle e, \rho, \sigma \rangle \Downarrow v_2$, allora $v_1 = v_2$
- **Type soundness**: se un'espressione è ben tipata, la sua valutazione non produce errori di tipo

Nota – Anticipazione: regole di tipo per gli operatori di confronto. Il sistema di tipi formale di MAO (presentato nel capitolo dedicato al linguaggio MAO completo) include una regola **T-Cop** per gli operatori di confronto ($<, \leq, >, \geq, ==, \neq$). In tale regola, gli operandi sono vincolati ad avere tipo `int`. Si osservi tuttavia che, a livello semantico, gli operatori di uguaglianza `==` e disuguaglianza `!=` sono definiti anche su operandi booleani (come indicato nella tabella degli operatori di MiniMao). Questa discrepanza è una scelta progettuale comune nei linguaggi didattici: il type checker può essere reso più permissivo estendendo T-Cop con una seconda variante che ammetta operandi di tipo `bool` per `==` e `!=`. Per i dettagli completi si rimanda alla sezione sulle regole di type checking nel capitolo su MAO.

3.4 Esercizi: Dimostrazioni Induttive su Grammatiche

In questa sezione vengono presentati esercizi di dimostrazione per induzione strutturale su grammatiche context-free. Per ogni esercizio si richiede di dimostrare una proprietà valida per tutte le stringhe del linguaggio generato.

3.4.1 Esercizio 1: Bilanciamento di a e b

Esempio 3.8 – Grammatica. Data la grammatica:

$$S ::= aSb \mid ab$$

Dimostrare per induzione strutturale che ogni stringa $w \in L(S)$ soddisfa $\#_a(w) = \#_b(w)$, dove $\#_a(w)$ indica il numero di occorrenze del simbolo a nella stringa w.

Dimostrazione. Procediamo per induzione strutturale sulla derivazione di w.

Caso base: $S \to ab$

La stringa generata è $w = ab$. Contiamo le occorrenze:
- $\#_a(ab) = 1$
- $\#_b(ab) = 1$

Quindi $\#_a(w) = \#_b(w) = 1$. ✓

Caso induttivo: $S \to aSb$

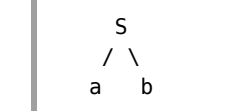
Assumiamo per ipotesi induttiva che la stringa w' generata da S soddisfi $\#_a(w') = \#_b(w') = k$ per qualche $k \geq 1$.

La nuova stringa è $w = aw'b$. Contiamo le occorrenze:
- $\#_a(aw'b) = 1 + \#_a(w') = 1 + k$
- $\#_b(aw'b) = \#_b(w') + 1 = k + 1$

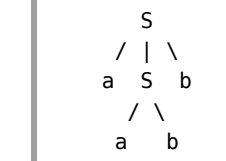
Quindi $\#_a(w) = \#_b(w) = k + 1$. ✓ ■

Esempio 3.9 – Alberi di derivazione. Mostriamo gli alberi di derivazione per alcune stringhe del linguaggio:

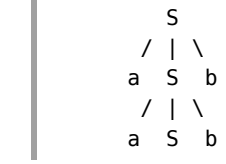
Stringa *ab* (caso base):



Stringa *aabb* (una applicazione della regola ricorsiva):



Stringa *aaabbb* (due applicazioni della regola ricorsiva):



```
    / \
   a   b
```

Nota. Il linguaggio $L(S) = \{a^n b^n \mid n \geq 1\}$ è l'esempio classico di linguaggio context-free non regolare. La proprietà dimostrata ($\#_a = \#_b$) è necessaria ma non sufficiente per caratterizzare il linguaggio (ad esempio *abab* ha lo stesso numero di a e b ma non appartiene a $L(S)$).

3.4.2 Esercizio 2: Stringhe con esattamente una c

Esempio 3.10 – Grammatica. Data la grammatica:

$$S ::= bSa \mid aSb \mid c$$

Dimostrare per induzione strutturale che ogni stringa $w \in L(S)$ soddisfa:
1. $\#_c(w) = 1$ (esattamente una occorrenza di c)
2. $\#_a(w) = \#_b(w)$ (stesso numero di a e b)

Dimostrazione. Procediamo per induzione strutturale sulla derivazione di w.

Caso base: $S \to c$

La stringa generata è $w = c$. Verifichiamo:
- $\#_c(c) = 1$ ✓
- $\#_a(c) = 0 = \#_b(c)$ ✓

Caso induttivo 1: $S \to bSa$

Assumiamo per ipotesi induttiva che la stringa w' generata da S soddisfi $\#_c(w') = 1$ e $\#_a(w') = \#_b(w') = k$ per qualche $k \geq 0$.

La nuova stringa è $w = bw'a$. Verifichiamo:
- $\#_c(bw'a) = \#_c(w') = 1$ ✓
- $\#_a(bw'a) = \#_a(w') + 1 = k + 1$
- $\#_b(bw'a) = 1 + \#_b(w') = 1 + k = k + 1$

Quindi $\#_a(w) = \#_b(w) = k + 1$. ✓

Caso induttivo 2: $S \to aSb$

Assumiamo per ipotesi induttiva che la stringa w' generata da S soddisfi $\#_c(w') = 1$ e $\#_a(w') = \#_b(w') = k$ per qualche $k \geq 0$.

La nuova stringa è $w = aw'b$. Verifichiamo:
- $\#_c(aw'b) = \#_c(w') = 1$ ✓
- $\#_a(aw'b) = 1 + \#_a(w') = 1 + k$
- $\#_b(aw'b) = \#_b(w') + 1 = k + 1$

Quindi $\#_a(w) = \#_b(w) = k + 1$. ✓ ■

Esempio 3.11 – Alberi di derivazione. Mostriamo gli alberi di derivazione per alcune stringhe del linguaggio:

Stringa *c* (caso base):

```
  S
  |
  c
```

Stringa *bca* (regola bSa):

```
    S
  / | \
 b  S  a
    |
    c
```

Stringa *acb* (regola aSb):

```
    S
  / | \
 a  S  b
    |
    c
```

Stringa *abcab* (composizione di regole):

```
        S
      / | \
     a  S  b
      / | \
     b  S  a
        |
        c
```

Derivazione: $S \to aSb \to abSab \to abcab$

Nota. Questa grammatica genera il linguaggio delle stringhe palindrome? No! Ad esempio *abcba* non è generabile. La grammatica genera stringhe dove c è sempre al centro, ma le a e b a sinistra e destra di c non sono necessariamente simmetriche.

3.4.3 Esercizio 3: Almeno una a

Esempio 3.12 – Grammatica. Data la grammatica:

$$S ::= Sa \mid Sb \mid a$$

Dimostrare per induzione strutturale che ogni stringa $w \in L(S)$ soddisfa $\#_a(w) \geq 1$.

Dimostrazione. Procediamo per induzione strutturale sulla derivazione di w.

Caso base: $S \to a$

La stringa generata è $w = a$. Verifichiamo:
- $\#_a(a) = 1 \geq 1$ ✓

Caso induttivo 1: $S \to Sa$

Assumiamo per ipotesi induttiva che la stringa w' generata da S soddisfi $\#_a(w') \geq 1$.

La nuova stringa è $w = w'a$. Verifichiamo:
- $\#_a(w'a) = \#_a(w') + 1 \geq 1 + 1 = 2 \geq 1$ ✓

Caso induttivo 2: $S \to Sb$

Assumiamo per ipotesi induttiva che la stringa w' generata da S soddisfi $\#_a(w') \geq 1$.

La nuova stringa è $w = w'b$. Verifichiamo:
- $\#_a(w'b) = \#_a(w') \geq 1$ ✓

(Aggiungere una b non cambia il numero di a, che rimane almeno 1.) ■

Esempio 3.13 – Alberi di derivazione. Mostriamo gli alberi di derivazione per alcune stringhe del linguaggio:

Stringa a (caso base):

```
S
|
a
```

Stringa ab:

```
  S
 / \
S   b
|
a
```

Stringa aa:

```
  S
 / \
S   a
|
a
```

Stringa $abab$:

```
        S
       / \
      S   b
     / \
    S   a
   / \
  S   b
  |
  a
```

Derivazione: $S \to Sb \to Sab \to Sbab \to abab$

Nota. Questa grammatica genera il linguaggio $L(S) = \{a\} \cdot (a \mid b)^*$, cioè tutte le stringhe su $\{a, b\}$ che iniziano con a. La proprietà dimostrata ($\#_a \geq 1$) è una conseguenza immediata di questa caratterizzazione: ogni stringa inizia con almeno una a.

3.4.4 Osservazioni metodologiche

Nota – Schema generale per l'induzione strutturale. Per dimostrare una proprietà $P(w)$ per tutte le stringhe $w \in L(S)$:

1. **Identificare i casi base**: produzioni della forma $X ::= \omega$ dove $\omega \in T^*$ (solo terminali)
2. **Identificare i casi induttivi**: produzioni della forma $X ::= \alpha_1 Y_1 \alpha_2 Y_2 ... Y_n \alpha_{n+1}$ dove Y_i sono non-terminali
3. **Per ogni caso base**: verificare direttamente che $P(\omega)$ vale

4. **Per ogni caso induttivo**: assumere che $P(w_i)$ valga per le stringhe w_i generate dai non-terminali Y_i (ipotesi induttiva), e dimostrare che $P(\alpha_1 w_1 \alpha_2 w_2 ... w_n \alpha_{n+1})$ vale

3.5 MiniMao

In questo capitolo introduciamo **MiniMao**, una versione ristretta del linguaggio MAO. L'obiettivo è definire formalmente la *sintassi* (quali programmi sono corretti) e la *semantica operazionale* (come i programmi vengono eseguiti) utilizzando i sistemi logici introdotti nella sezione precedente.

3.5.1 Variabili, ambiente e memoria

La programmazione consiste nell'ideare uno o più algoritmi che risolvano un problema, valutarne l'efficacia e codificarli in un linguaggio eseguibile da un calcolatore. Un programma è una sequenza di istruzioni che indicano al calcolatore le operazioni da eseguire. Come istruzione elementare consideriamo la possibilità di memorizzare il risultato di un calcolo. Per riferirsi a questi valori si utilizzano nomi simbolici detti nomi di variabile.

Nota – Equazioni vs assegnamenti. Equazioni matematiche e assegnamenti sono due concetti profondamente diversi. Consideriamo $n = n^2 - 2$:

- Se interpretato come **equazione matematica**, cerchiamo il valore di n che verifica l'uguaglianza: le soluzioni sono $n \in \{2, -1\}$.
- Se interpretato come **assegnamento**, il significato è operazionale: se n attualmente vale 5, dopo l'assegnamento n assume il valore $5^2 - 2 = 23$.

Per evitare ambiguità, MAO usa simboli diversi: `=` per la dichiarazione e `:=` per l'assegnamento.

3.5.1.1 Concetto di variabile

Una variabile in MAO è modellata come una «scatola» dotata di un **nome** (l'identificatore), un **tipo** (che determina quali valori può contenere) e un **valore corrente** (il contenuto della scatola). La dichiarazione crea la variabile e le assegna un valore iniziale:

```
int eta = 15;
```

Successivamente, il valore può cambiare tramite assegnamento (si noti l'uso di `:=`):

```
eta := 16;
```

3.5.1.2 Stato del programma

Durante l'esecuzione di un programma possono esistere molte variabili, ciascuna con il proprio valore. Lo stato del programma è l'insieme di tutte le variabili e dei loro valori in un dato istante dell'esecuzione.

Esempio 3.14. Uno stato con tre variabili:

```
{eta = 16, studente = true, nome = "Jacob"}
```

3.5.2 Sintassi di MiniMao

Introduciamo ora la sintassi di MiniMao, la versione ristretta del linguaggio MAO. Prima definiamo la sintassi usando grammatiche formali; in un secondo momento definiremo la semantica utilizzando sistemi logici.

Le **categorie sintattiche** di MiniMao sono:

Definizione 3.10 – Categorie sintattiche.
- **Valori (V)**: dati elementari come numeri interi o booleani
- **Identificatori (Id)**: simboli per riferirsi a variabili
- **Espressioni (E)**: combinano valori, identificatori e operatori per produrre nuovi valori
- **Tipi (T)**: indicano l'insieme dei valori ammissibili per una variabile
- **Comandi (C)**: descrivono le azioni da eseguire (modificano lo stato)

Definizione 3.11 – Valori V. Inizialmente consideriamo solo programmi che manipolano valori interi ($\mathbb{Z}$) e booleani ($\mathbb{B}$):

$$V ::= n \mid \text{true} \mid \text{false}$$

dove n indica un qualsiasi numero intero.

Definizione 3.12 – Identificatori Id. Gli identificatori sono nomi simbolici che permettono al programmatore di riferirsi in modo chiaro e univoco a variabili. In MAO sono stringhe alfanumeriche che possono contenere il carattere underscore (_).

Esempio 3.15. `mio_id1`, `x`, `eta`, `contatore`

Nota. Alcune parole chiave del linguaggio (come `if`, `while`, `int`, `bool`, `true`, `false`, `skip`) sono riservate e non possono essere usate come identificatori.

Definizione 3.13 – Espressioni E. Le espressioni combinano valori, identificatori e operatori per produrre un nuovo valore (intero o booleano):
$E ::= V \mid \text{Id} \mid E \;\; \text{bop} \;\; E \mid \text{uop} \;\; E \mid (E)$

dove `bop` indica un operatore binario e `uop` un operatore unario.

Definizione 3.14 – Operatori binari. Gli operatori binari prendono due operandi e producono un risultato:

Simbolo	Nome	Tipo operandi	Tipo risultato
$+$	addizione	$\mathbb{Z} \times \mathbb{Z}$	$\mathbb{Z}$
$-$	sottrazione	$\mathbb{Z} \times \mathbb{Z}$	$\mathbb{Z}$
$\times$	moltiplicazione	$\mathbb{Z} \times \mathbb{Z}$	$\mathbb{Z}$
$\div$	divisione intera	$\mathbb{Z} \times \mathbb{Z}_{\neq 0}$	$\mathbb{Z}$
mod	resto (modulo)	$\mathbb{Z} \times \mathbb{Z}_{\neq 0}$	$\mathbb{Z}$
$<$	minore	$\mathbb{Z} \times \mathbb{Z}$	$\mathbb{B}$
$\leq$	minore o uguale	$\mathbb{Z} \times \mathbb{Z}$	$\mathbb{B}$
$>$	maggiore	$\mathbb{Z} \times \mathbb{Z}$	$\mathbb{B}$
$\geq$	maggiore o uguale	$\mathbb{Z} \times \mathbb{Z}$	$\mathbb{B}$
$==$	uguaglianza	$\mathbb{Z} \times \mathbb{Z}$ o $\mathbb{B} \times \mathbb{B}$	$\mathbb{B}$
$\neq$	disuguaglianza	$\mathbb{Z} \times \mathbb{Z}$ o $\mathbb{B} \times \mathbb{B}$	$\mathbb{B}$
$\wedge$	and logico	$\mathbb{B} \times \mathbb{B}$	$\mathbb{B}$
$\vee$	or logico	$\mathbb{B} \times \mathbb{B}$	$\mathbb{B}$

Tabella 5: Operatori binari in MiniMao

Nota. La divisione intera $\div$ e il modulo mod richiedono che il secondo operando sia diverso da zero; l'applicazione a zero produce un errore.

Definizione 3.15 – Operatori unari. Gli operatori unari prendono un solo operando:

Simbolo	Nome	Tipo operando	Tipo risultato
$-$	negazione aritmetica	$\mathbb{Z}$	$\mathbb{Z}$
$\neg$	negazione logica	$\mathbb{B}$	$\mathbb{B}$

Tabella 6: Operatori unari in MiniMao

Definizione 3.16 – Tipi T. Un tipo può essere un tipo base o un tipo composto. Per ogni tipo base assumiamo un valore di default (0 per interi e `false` per booleani):
$T ::= T_b \mid T_c$
dove $T_b \in \{\text{int}, \text{bool}\}$ sono tipi base.

Definizione 3.17 – Comandi C. I comandi descrivono le azioni che il programma deve eseguire:
$C ::= \text{skip;} \mid T\ \ \text{Id} = E; \mid \text{Id} := E; \mid CC \mid \{C\} \mid \text{if}(E)\{C\}\ \text{else}\ \{C\} \mid \text{while}(E)\{C\}$

In dettaglio:

- `skip;` – comando vuoto (non fa nulla)
- `T Id = E;` – dichiarazione di variabile con inizializzazione
- `Id := E;` – assegnamento a variabile esistente
- `C C` – composizione sequenziale di due comandi
- `{C}` – blocco (introduce uno scope)
- `if(E){C}else{C}` – comando condizionale
- `while(E){C}` – ciclo iterativo

Esempio 3.16 – Programma in MiniMao.

```
int x = 1;
int n = 0;
while(x <= y) {
    x := x * 2;
    n := n + 1;
}
```

Si noti l'uso di `=` per dichiarare variabili e `:=` per gli assegnamenti.

3.6 Semantica di MiniMao

La sintassi ci dice *quali* programmi sono corretti; la **semantica** ci dice *cosa fanno*. La semantica è lo strumento che ci permette di ragionare formalmente sul comportamento di un programma e quindi di studiare proprietà come correttezza, equivalenza, terminazione e divergenza.

Definizione 3.18 – Semantica operazionale. La **semantica operazionale** descrive il comportamento di un programma in termini dei passi che un'opportuna macchina astratta compie per eseguirlo.

Definizione 3.19 – Macchina astratta. Una **macchina astratta** è una macchina semplificata ideale il cui stato è dato da due componenti: l'**ambiente** ρ e la **memoria** σ.

3.6.1 Ambiente e memoria

L'**ambiente** $\rho : \mathbb{I} \hookrightarrow \mathbb{L}$ è una funzione parziale che associa identificatori a locazioni di memoria. Risponde alla domanda: «dove si trova la variabile x?»

La **memoria** $\sigma : \mathbb{L} \hookrightarrow \mathbb{V}$ è una funzione parziale che associa locazioni a valori. Risponde alla domanda: «che valore contiene la locazione l?»

Per leggere il valore di una variabile x servono due passi: prima si cerca la locazione $l = \rho(x)$, poi si legge il valore $v = \sigma(l)$.

Esempio 3.17 – Ambiente e memoria. Con due variabili `eta` e `studente`:

$\rho(\text{eta}) = l_0,\ \rho(\text{studente}) = l_1 \qquad \sigma(l_0) = 15,\ \sigma(l_1) = \text{true}$

La variabile `eta` si trova nella locazione l_0, che contiene il valore 15; la variabile `studente` si trova in l_1, che contiene `true`.

Nota. Una funzione è *parziale* quando potrebbe non essere definita per tutti gli argomenti del dominio. Ad esempio, $\rho(\text{z})$ non è definita se la variabile `z` non è stata dichiarata.

3.6.2 Stato del programma

Uno **stato** della macchina astratta è una coppia ambiente-memoria $s = (\rho, \sigma)$.

Quando le funzioni parziali sono definite su un numero finito di argomenti, le rappresentiamo elencando tutte le associazioni argomento-valore:

$$\rho = [x \mapsto l_1, y \mapsto l_2],\ \sigma = [l_1 \mapsto 15, l_2 \mapsto \text{true}]$$

La notazione $\rho[x \mapsto l]$ indica l'ambiente ottenuto da ρ aggiungendo (o sovrascrivendo) l'associazione $x \mapsto l$. Analogamente $\sigma[l \mapsto v]$ per la memoria.

3.6.3 Predicati semantici

Per definire la semantica dei programmi abbiamo bisogno di specificare formalmente come si valutano le espressioni e come si eseguono i comandi. Introduciamo due famiglie di *giudizi* (predicati semantici):

> **Definizione 3.20 – Giudizio di valutazione delle espressioni.** $\langle E, \rho, \sigma \rangle \Downarrow v$
>
> Si legge: «l'espressione E, nello stato (ρ, σ), ha valore v». L'espressione viene valutata ma lo stato **non cambia** (espressioni pure).

> **Definizione 3.21 – Giudizio di esecuzione dei comandi.** $\langle C, \rho, \sigma \rangle \to \langle \rho', \sigma' \rangle$
>
> Si legge: «l'esecuzione del comando C nello stato iniziale (ρ, σ) termina producendo lo stato finale (ρ', σ')». Il comando può modificare sia l'ambiente (aggiungendo nuove variabili) che la memoria (cambiando valori).

3.6.4 Espressioni pure e con effetti collaterali

Le espressioni con effetti collaterali sono espressioni la cui valutazione può comportare modifiche alla memoria (come allocazione o modifica di celle). In tal caso la valutazione si esprime come $\langle E, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$.

Le espressioni **pure** non alterano lo stato: la valutazione produce un valore ma lascia ambiente e memoria invariati. MiniMao ammette solo espressioni pure, il che semplifica notevolmente le regole semantiche.

3.6.5 Regole semantiche per le espressioni

Le regole seguenti definiscono formalmente come valutare ciascun tipo di espressione. Per ogni regola, le premesse (sopra la linea) specificano le condizioni necessarie, e la conclusione (sotto la linea) specifica il risultato della valutazione.

3.6.5.1 Valori letterali

I valori letterali (numeri e booleani) si valutano a se stessi. Sono assiomi perché non richiedono premesse:

$$\frac{}{\langle n, \rho, \sigma \rangle \Downarrow n} \quad \text{(Val-Int)}$$

$$\frac{}{\langle \text{true}, \rho, \sigma \rangle \Downarrow \text{true}} \quad \text{(Val-True)} \qquad \frac{}{\langle \text{false}, \rho, \sigma \rangle \Downarrow \text{false}} \quad \text{(Val-False)}$$

> **Nota.** Nelle regole (Val-Int), (Val-True), (Val-False), l'ambiente ρ e la memoria σ compaiono ma non vengono utilizzati. Il valore di un letterale non dipende dallo stato.

3.6.5.2 Variabili

Per valutare una variabile si effettua una doppia ricerca: prima nell'ambiente (per trovare la locazione), poi nella memoria (per trovare il valore):

$$\frac{\rho(\text{Id}) = l \qquad \sigma(l) = v}{\langle \text{Id}, \rho, \sigma \rangle \Downarrow v} \qquad \text{(Val-Var)}$$

La regola (Val-Var) ha due premesse: $\rho(\text{Id}) = l$ (l'identificatore è associato alla locazione l nell'ambiente) e $\sigma(l) = v$ (la locazione l contiene il valore v nella memoria).

3.6.5.3 Operatori binari e unari

$$\frac{\langle E_1, \rho, \sigma \rangle \Downarrow v_1 \quad \langle E_2, \rho, \sigma \rangle \Downarrow v_2 \quad v = v_1 \text{ bop } v_2}{\langle E_1 \text{ bop } E_2, \rho, \sigma \rangle \Downarrow v} \qquad \text{(Val-Bop)}$$

La regola (Val-Bop) si legge: «per valutare E_1 bop E_2, si valutano prima E_1 e E_2 separatamente (ottenendo v_1 e v_2), poi si applica l'operatore `bop` ai risultati».

$$\frac{\langle E, \rho, \sigma \rangle \Downarrow v' \qquad v = \text{uop } v'}{\langle \text{uop } E, \rho, \sigma \rangle \Downarrow v} \qquad \text{(Val-Uop)}$$

La regola (Val-Uop) è analoga ma con un solo operando.

3.6.5.4 Espressioni parentesizzate

> **Nota – Parentesi nelle espressioni.** La grammatica delle espressioni include la produzione (E), che permette di racchiudere un'espressione tra parentesi tonde. Tuttavia **non è necessaria una regola semantica separata** per le espressioni parentesizzate: le parentesi servono esclusivamente a disambiguare l'ordine di valutazione a livello sintattico (ad esempio, per forzare $(a + b) * c$ invece di $a + (b * c)$). Una volta costruito l'albero sintattico, la struttura delle sotto-espressioni è completamente determinata e le parentesi non influiscono sulla valutazione. In altre parole, valutare (E) equivale esattamente a valutare E: le regole (Val-Bop) e (Val-Uop) si applicano direttamente alla struttura dell'espressione contenuta.

3.7 Regole semantiche per i comandi

I comandi, a differenza delle espressioni, **modificano lo stato**. Le regole seguenti descrivono come ciascun tipo di comando trasforma la coppia (ρ, σ).

3.7.1 Dichiarazione, assegnamento e sequenza

Per comprendere le regole formali, descriviamo prima informalmente ciascuna operazione.

Dichiarazione (`T Id = E;`): per eseguire una dichiarazione nello stato (ρ, σ) si deve:
1. Valutare l'espressione E nello stato corrente, ottenendo il valore v.
2. Trovare una locazione di memoria l non ancora usata ($l \notin \text{dom}(\sigma)$).
3. Estendere la memoria σ con l'associazione $l \mapsto v$.
4. Estendere l'ambiente ρ con l'associazione $\text{Id} \mapsto l$.

Assegnamento (`Id := E;`): per eseguire un assegnamento nello stato (ρ, σ) si deve:
1. Valutare l'espressione E nello stato corrente, ottenendo il valore v.
2. Individuare la locazione l che l'ambiente ρ associa all'identificatore Id.
3. Aggiornare la memoria σ con l'associazione $l \mapsto v$ (il vecchio valore viene sovrascritto).
4. L'ambiente rimane invariato (non si creano nuove variabili).

Composizione sequenziale (`C1 C2`): per eseguire due comandi in sequenza:
1. Eseguire C_1 nello stato (ρ, σ), ottenendo lo stato (ρ_1, σ_1).
2. Eseguire C_2 nello stato (ρ_1, σ_1), ottenendo lo stato finale (ρ_2, σ_2).

3.7.2 Skip

Il comando `skip;` non modifica lo stato: è un assioma (nessuna premessa).

$$\langle \text{skip};, \rho, \sigma \rangle \to \langle \rho, \overline{\sigma} \rangle \qquad \text{(Cmd-Skip)}$$

3.7.3 Dichiarazione

$$\frac{\langle E, \rho, \sigma \rangle \Downarrow v \quad l \notin \text{dom}(\sigma)}{\langle T \;\; \text{Id} = E;, \rho, \sigma \rangle \to \langle \rho[\text{Id} \mapsto l], \sigma[l \mapsto v] \rangle} \qquad \text{(Cmd-Decl)}$$

La regola (Cmd-Decl) modifica **sia** l'ambiente (aggiungendo $\text{Id} \mapsto l$) **sia** la memoria (aggiungendo $l \mapsto v$). La premessa $l \notin \text{dom}(\sigma)$ garantisce che la locazione scelta sia fresca (non già in uso).

3.7.4 Assegnamento

$$\frac{\langle E, \rho, \sigma \rangle \Downarrow v \quad \rho(\text{Id}) = l}{\langle \text{Id} := E;, \rho, \sigma \rangle \to \langle \rho, \sigma[l \mapsto v] \rangle} \qquad \text{(Cmd-Assign)}$$

La regola (Cmd-Assign) modifica **solo** la memoria (aggiornando il valore nella locazione l). L'ambiente rimane invariato: ρ compare identico nello stato iniziale e finale.

3.7.5 Sequenza

$$\frac{\langle C_1, \rho, \sigma \rangle \to \langle \rho_1, \sigma_1 \rangle \quad \langle C_2, \rho_1, \sigma_1 \rangle \to \langle \rho_2, \sigma_2 \rangle}{\langle C_1 C_2, \rho, \sigma \rangle \to \langle \rho_2, \sigma_2 \rangle} \qquad \text{(Cmd-Seq)}$$

La regola (Cmd-Seq) mostra chiaramente il «passaggio di stato»: lo stato prodotto da C_1 diventa lo stato iniziale di C_2.

3.7.6 Sviluppo sequenziale

Le derivazioni ad albero possono diventare complesse e difficili da leggere per programmi con molti comandi. Lo **sviluppo sequenziale** è una notazione alternativa più compatta che descrive l'esecuzione come una sequenza di stati, uno per ogni comando eseguito.

> **Definizione 3.22 – Sviluppo sequenziale.** Lo sviluppo sequenziale rappresenta l'esecuzione di un programma come una sequenza verticale che alterna stati e comandi:
>
> $$\begin{gathered} \langle C, \rho_0, \sigma_0 \rangle \\ \downarrow \\ \langle \rho_1, \sigma_1 \rangle \text{ (dopo primo comando)} \\ \downarrow \\ \langle \rho_2, \sigma_2 \rangle \text{ (dopo secondo comando)} \\ \ldots \\ \downarrow \\ \langle \rho_n, \sigma_n \rangle \text{ (stato finale)} \end{gathered}$$

Notazione semplificata: invece di scrivere la configurazione completa con ambiente e memoria separati, spesso «fondiamo» le due componenti e scriviamo lo stato come un insieme di associazioni variabile $\mapsto$ valore:

$$\{x \mapsto v_1, y \mapsto v_2, ...\}$$

Questa notazione nasconde le locazioni di memoria e si legge: «la variabile x ha valore v_1, la variabile y ha valore v_2, …».

Esempio 3.18 – Derivazione vs sviluppo sequenziale. Consideriamo `C1 = y := x;` e `C2 = x := y;` con stato iniziale $\rho = [x \mapsto l_1, y \mapsto l_2]$, $\sigma = [l_1 \mapsto 5, l_2 \mapsto 3]$.

Come derivazione (albero): l'albero di derivazione ha struttura annidata e richiede di leggere dal basso verso l'alto.

$$\dfrac{\dfrac{}{\langle x, \rho, \sigma\rangle \Downarrow 5} \quad \dfrac{\langle y := x;, \rho, \sigma\rangle \to \langle \rho, \sigma[l_2 \mapsto 5]\rangle \qquad \dfrac{}{\langle y, \rho, \sigma'\rangle \Downarrow 5}}{\langle x := y;, \rho, \sigma'\rangle \to \langle \rho, \sigma'[l_1 \mapsto 5]\rangle}}{\langle \text{y:=x; x:=y;}, \rho, \sigma\rangle \to \langle \rho, \sigma''\rangle}$$

Come sviluppo sequenziale (lineare): molto più leggibile.

$$\begin{gathered}\{x \mapsto 5, y \mapsto 3\} \\ \downarrow \texttt{y := x;} \\ \{x \mapsto 5, y \mapsto 5\} \\ \downarrow \texttt{x := y;} \\ \{x \mapsto 5, y \mapsto 5\}\end{gathered}$$

Si noti che il programma **non scambia** i valori di x e y: dopo `y := x;`, entrambe le variabili valgono 5, quindi `x := y;` assegna 5 a x (che già vale 5).

Esempio 3.19 – Sviluppo sequenziale completo. Consideriamo il programma:

```
int x = 3;
int y = 0;
y := x + 1;
x := y * 2;
```

Stato iniziale: $\rho_0 = \emptyset$, $\sigma_0 = \emptyset$

Passo 1: `int x = 3;` (regola Cmd-Decl)
- Valuto l'espressione: $\langle 3, \rho_0, \sigma_0\rangle \Downarrow 3$
- Alloco una nuova locazione: $l_1 \notin \text{dom}(\sigma_0)$
- Estendo ambiente: $\rho_1 = \rho_0[x \mapsto l_1] = [x \mapsto l_1]$
- Estendo memoria: $\sigma_1 = \sigma_0[l_1 \mapsto 3] = [l_1 \mapsto 3]$

Passo 2: `int y = 0;` (regola Cmd-Decl)
- Valuto: $\langle 0, \rho_1, \sigma_1\rangle \Downarrow 0$
- Alloco: $l_2 \notin \text{dom}(\sigma_1)$
- $\rho_2 = \rho_1[y \mapsto l_2] = [x \mapsto l_1, y \mapsto l_2]$
- $\sigma_2 = \sigma_1[l_2 \mapsto 0] = [l_1 \mapsto 3, l_2 \mapsto 0]$

Passo 3: `y := x + 1;` (regola Cmd-Assign)
- Valuto $x + 1$: $\langle x + 1, \rho_2, \sigma_2\rangle \Downarrow 3 + 1 = 4$
- Trovo la locazione di y: $\rho_2(y) = l_2$
- Aggiorno memoria: $\sigma_3 = \sigma_2[l_2 \mapsto 4] = [l_1 \mapsto 3, l_2 \mapsto 4]$

- L'ambiente resta: $\rho_3 = \rho_2$

Passo 4: `x := y * 2;` (regola Cmd-Assign)
- Valuto $y * 2$: $\langle y * 2, \rho_3, \sigma_3 \rangle \Downarrow 4 * 2 = 8$
- Trovo la locazione di x: $\rho_3(x) = l_1$
- Aggiorno memoria: $\sigma_4 = \sigma_3[l_1 \mapsto 8] = [l_1 \mapsto 8, l_2 \mapsto 4]$

Stato finale: $\rho_4 = [x \mapsto l_1, y \mapsto l_2]$, $\sigma_4 = [l_1 \mapsto 8, l_2 \mapsto 4]$

Quindi $x = 8$ e $y = 4$.

3.7.7 Blocchi e scoping

3.7.7.1 Il blocco {C}

Un **blocco** è un comando racchiuso tra parentesi graffe `{C}`. Tutte le variabili dichiarate all'interno del blocco sono visibili solo al suo interno. Lo scope (ambito di visibilità) di una variabile definisce la porzione di codice nella quale la variabile può essere dichiarata, letta o modificata.

In MAO una variabile dichiarata all'interno di un blocco è visibile:
- In tutti i comandi successivi alla dichiarazione, nello stesso blocco.
- All'interno dei blocchi annidati.

All'uscita dal blocco, le variabili dichiarate al suo interno cessano di essere visibili.

3.7.7.2 Shadowing

È possibile dichiarare una variabile all'interno di un blocco con lo stesso nome di una variabile già dichiarata in un blocco esterno. In questo caso la variabile interna **nasconde** (shadow) quella esterna: all'interno del blocco, il nome si riferisce alla nuova variabile; all'uscita dal blocco, il nome torna a riferirsi alla variabile originale.

3.7.8 Blocco

$$\frac{\langle C, \rho, \sigma \rangle \to \langle \rho', \sigma' \rangle}{\langle \{C\}, \rho, \sigma \rangle \to \langle \rho, \sigma' \rangle} \quad \text{(Cmd-Block)}$$

Nota – Semantica del blocco. La regola (Cmd-Block) è cruciale: lo stato finale del blocco ha l'**ambiente originale** ρ (non ρ'), ma la **memoria modificata** σ'. Questo significa che:
- Le variabili dichiarate nel blocco vengono «dimenticate» (l'ambiente torna quello di prima).
- Le modifiche alla memoria persistono (se il blocco ha modificato il valore di una variabile esterna, la modifica rimane).
- Le locazioni allocate nel blocco restano in memoria ma diventano inaccessibili (nessun nome punta più ad esse).

3.7.9 Comando condizionale (if-then-else)

I comandi condizionali servono per prendere decisioni durante l'esecuzione. L'espressione E (detta **guardia**) viene valutata: se produce `true`, si esegue il ramo «then» (C_1); se produce `false`, si esegue il ramo «else» (C_2).

```
if (E) {C1} else {C2}
```

Esempio 3.20 – Comando condizionale in MAO.

```
if (piove) {
    ombrello := true;
} else {
    ombrello := false;
}
```

Le regole semantiche distinguono i due casi:

$$\frac{\langle E, \rho, \sigma\rangle \Downarrow \text{true} \quad \langle C_1, \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \text{if}(E)\{C_1\}\text{else}\{C_2\}, \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle} \quad \text{(Cmd-IfTrue)}$$

$$\frac{\langle E, \rho, \sigma\rangle \Downarrow \text{false} \quad \langle C_2, \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle}{\langle \text{if}(E)\{C_1\}\text{else}\{C_2\}, \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle} \quad \text{(Cmd-IfFalse)}$$

3.7.10 Comando if-then (senza else)

Il comando `if(E){C}` senza ramo `else` si esprime in MAO come un `if-else` con `skip;` nel ramo falso:

```
if (E) {C1} else {skip;}
```

Esempio 3.21 – Comando condizionale senza else.

```
if (piove) {
    ombrello := true;
}
```

Equivale a: `if (piove) { ombrello := true; } else { skip; }`

Nota. Le parentesi tonde e graffe svolgono due compiti distinti: le parentesi tonde `()` delimitano la guardia e fissano l'ordine di valutazione nelle espressioni; le parentesi graffe `{}` delimitano un blocco di comandi e introducono uno scope.

Esempio 3.22 – Sviluppo sequenziale con condizionale. Consideriamo:

```
int a = 5;
int b = 3;
int max = 0;
if(a > b){
    max := a;
} else {
    max := b;
}
```

Dopo le dichiarazioni: $\rho = [a \mapsto l_1, b \mapsto l_2, \max \mapsto l_3]$, $\sigma = [l_1 \mapsto 5, l_2 \mapsto 3, l_3 \mapsto 0]$

Valutazione della guardia: $\langle a > b, \rho, \sigma\rangle \Downarrow \langle 5 > 3\rangle \Downarrow \text{true}$

Poiché la guardia è vera, si applica (Cmd-IfTrue) e si esegue il ramo `then`:

Esecuzione `max := a;`:
- Valuto a: $\sigma(\rho(a)) = \sigma(l_1) = 5$
- Aggiorno: $\sigma' = \sigma[l_3 \mapsto 5] = [l_1 \mapsto 5, l_2 \mapsto 3, l_3 \mapsto 5]$

Stato finale: $\max = 5$, che è corretto poiché $\max(5, 3) = 5$.

3.7.11 Cicli

3.7.11.1 Comando iterativo (while)

Il ciclo `while` ripete l'esecuzione di un corpo di comandi finché la guardia è vera. È l'unico costrutto iterativo presente in MAO.

```
while (E) {C}
```

L'espressione E è una guardia booleana: se vale `true`, viene eseguito il corpo C e poi si ripete il controllo della guardia; se vale `false`, il ciclo termina e l'esecuzione prosegue con il comando successivo.

> **Esempio 3.23 – Comando iterativo in MAO.**
>
> ```
> while(cioccolatini > 0) {
>     cioccolatini := cioccolatini - 1;
> }
> ```

3.7.11.2 Semantica del while

Il while richiede due regole: una per il caso in cui la guardia è falsa (il ciclo termina) e una per il caso in cui è vera (il ciclo prosegue).

$$\frac{\langle E, \rho, \sigma\rangle \Downarrow \text{false}}{\langle \text{while}(E)\{C\}, \rho, \sigma\rangle \to \langle \rho, \sigma\rangle} \quad \text{(Cmd-WhileFalse)}$$

Se la guardia è falsa, il ciclo termina immediatamente: lo stato non cambia.

$$\frac{\langle E, \rho, \sigma\rangle \Downarrow \text{true} \quad \langle C, \rho, \sigma\rangle \to \langle \rho', \sigma'\rangle \quad \langle \text{while}(E)\{C\}, \rho', \sigma'\rangle \to \langle \rho'', \sigma''\rangle}{\langle \text{while}(E)\{C\}, \rho, \sigma\rangle \to \langle \rho'', \sigma''\rangle} \quad \text{(Cmd-WhileTrue)}$$

> **Nota – Ricorsività della regola WhileTrue.** La regola (Cmd-WhileTrue) è **ricorsiva**: tra le premesse compare di nuovo il giudizio `while(E){C}`, ma su uno stato aggiornato (ρ', σ'). Il ciclo viene quindi «srotolato» un passo alla volta: si esegue il corpo una volta, e poi si riesegue l'intero while sullo stato risultante.

3.8 Riepilogo ed esempi

Comando	Regole
`skip;`	(Cmd-Skip)
`T Id = E;`	(Cmd-Decl)
`Id := E;`	(Cmd-Assign)
`C1 C2`	(Cmd-Seq)
`{C}`	(Cmd-Block)
`if(E){C1}else{C2}`	(Cmd-IfTrue), (Cmd-IfFalse)
`while(E){C}`	(Cmd-WhileFalse), (Cmd-WhileTrue)

Tabella 7: Riepilogo delle regole semantiche di MiniMao

3.8.1 Esempi di sviluppo sequenziale

3.8.1.1 Ciclo while: somma dei primi n

Esempio 3.24 – Sviluppo sequenziale con ciclo while. Consideriamo:

```
int n = 3;
int s = 0;
while(n > 0){
    s := s + n;
    n := n - 1;
}
```

Stato iniziale dopo le dichiarazioni: $\rho = [n \mapsto l_1, s \mapsto l_2]$, $\sigma_0 = [l_1 \mapsto 3, l_2 \mapsto 0]$

Iterazione 1 (regola Cmd-WhileTrue):
- Guardia: $\langle n > 0, \rho, \sigma_0 \rangle \Downarrow 3 > 0 = \text{true}$
- `s := s + n;`: $s = 0 + 3 = 3$ — $\sigma_1 = [l_1 \mapsto 3, l_2 \mapsto 3]$
- `n := n - 1;`: $n = 3 - 1 = 2$ — $\sigma_{1'} = [l_1 \mapsto 2, l_2 \mapsto 3]$

Iterazione 2 (regola Cmd-WhileTrue):
- Guardia: $2 > 0 = \text{true}$
- `s := s + n;`: $s = 3 + 2 = 5$ — $\sigma_2 = [l_1 \mapsto 2, l_2 \mapsto 5]$
- `n := n - 1;`: $n = 2 - 1 = 1$ — $\sigma_{2'} = [l_1 \mapsto 1, l_2 \mapsto 5]$

Iterazione 3 (regola Cmd-WhileTrue):
- Guardia: $1 > 0 = \text{true}$
- `s := s + n;`: $s = 5 + 1 = 6$ — $\sigma_3 = [l_1 \mapsto 1, l_2 \mapsto 6]$
- `n := n - 1;`: $n = 1 - 1 = 0$ — $\sigma_{3'} = [l_1 \mapsto 0, l_2 \mapsto 6]$

Uscita dal ciclo (regola Cmd-WhileFalse):
- Guardia: $0 > 0 = \text{false}$ — il ciclo termina

Stato finale: $n = 0$, $s = 6$ (somma dei primi 3 naturali: $3 + 2 + 1 = 6$).

3.8.1.2 Ciclo while: calcolo del fattoriale

Esempio 3.25 – Sviluppo sequenziale: calcolo del fattoriale. Consideriamo il programma che calcola il fattoriale di 4:

```
int n = 4;
int f = 1;
while (n > 0) {
  f := f * n;
  n := n - 1;
}
```

Stato iniziale dopo le dichiarazioni:
- $\rho = [n \mapsto l_n, f \mapsto l_f]$
- $\sigma_0 = [l_n \mapsto 4, l_f \mapsto 1]$

Iterazione 1:
- Guardia: $\langle n > 0, \rho, \sigma_0 \rangle \Downarrow 4 > 0 = \text{true}$
- `f := f * n;`: $\langle f * n, \rho, \sigma_0 \rangle \Downarrow 1 \times 4 = 4$
  - $\sigma_{0'} = \sigma_0[l_f \mapsto 4] = [l_n \mapsto 4, l_f \mapsto 4]$
- `n := n - 1;`: $\langle n - 1, \rho, \sigma_{0'} \rangle \Downarrow 4 - 1 = 3$
  - $\sigma_1 = \sigma_{0'}[l_n \mapsto 3] = [l_n \mapsto 3, l_f \mapsto 4]$

Iterazione 2:
- Guardia: $\langle n > 0, \rho, \sigma_1 \rangle \Downarrow 3 > 0 = \text{true}$

- `f := f * n;`: $\langle f * n, \rho, \sigma_1 \rangle \Downarrow 4 \times 3 = 12$
  - $\sigma_{1'} = \sigma_1[l_f \mapsto 12] = [l_n \mapsto 3, l_f \mapsto 12]$
- `n := n - 1;`: $\langle n - 1, \rho, \sigma_{1'} \rangle \Downarrow 3 - 1 = 2$
  - $\sigma_2 = \sigma_{1'}[l_n \mapsto 2] = [l_n \mapsto 2, l_f \mapsto 12]$

Iterazione 3:
- Guardia: $\langle n > 0, \rho, \sigma_2 \rangle \Downarrow 2 > 0 = \text{true}$
- `f := f * n;`: $\langle f * n, \rho, \sigma_2 \rangle \Downarrow 12 \times 2 = 24$
  - $\sigma_{2'} = \sigma_2[l_f \mapsto 24] = [l_n \mapsto 2, l_f \mapsto 24]$
- `n := n - 1;`: $\langle n - 1, \rho, \sigma_{2'} \rangle \Downarrow 2 - 1 = 1$
  - $\sigma_3 = \sigma_{2'}[l_n \mapsto 1] = [l_n \mapsto 1, l_f \mapsto 24]$

Iterazione 4:
- Guardia: $\langle n > 0, \rho, \sigma_3 \rangle \Downarrow 1 > 0 = \text{true}$
- `f := f * n;`: $\langle f * n, \rho, \sigma_3 \rangle \Downarrow 24 \times 1 = 24$
  - $\sigma_{3'} = \sigma_3[l_f \mapsto 24] = [l_n \mapsto 1, l_f \mapsto 24]$
- `n := n - 1;`: $\langle n - 1, \rho, \sigma_{3'} \rangle \Downarrow 1 - 1 = 0$
  - $\sigma_4 = \sigma_{3'}[l_n \mapsto 0] = [l_n \mapsto 0, l_f \mapsto 24]$

Uscita dal ciclo:
- Guardia: $\langle n > 0, \rho, \sigma_4 \rangle \Downarrow 0 > 0 = \text{false}$ — il ciclo termina

Riepilogo delle iterazioni:

Iterazione	n	f	Guardia
Inizio	4	1	-
1	3	4	true
2	2	12	true
3	1	24	true
4	0	24	true
Fine	0	24	false

Tabella 8: Evoluzione dello stato nel calcolo di 4!

Stato finale: $n = 0$, $f = 24$ (infatti $4! = 4 \times 3 \times 2 \times 1 = 24$).

3.8.1.3 Blocchi e shadowing

Esempio 3.26 – Sviluppo sequenziale: blocchi e shadowing. Consideriamo il programma che illustra lo shadowing delle variabili:

```
int x = 10;
{
  int x = 5;
  x := x + 1;
}
x := x * 2;
```

Stato iniziale: $\rho_0 = \emptyset$, $\sigma_0 = \emptyset$

Passo 1: `int x = 10;` (dichiarazione esterna)
- Valuto: $\langle 10, \rho_0, \sigma_0 \rangle \Downarrow 10$
- Alloco nuova locazione: $l_1 \notin \text{dom}(\sigma_0)$
- $\rho_1 = [x \mapsto l_1]$
- $\sigma_1 = [l_1 \mapsto 10]$

Passo 2: Ingresso nel blocco `{ ... }`

- L'ambiente corrente è: $\rho_1 = [x \mapsto l_1]$
- La memoria corrente è: $\sigma_1 = [l_1 \mapsto 10]$

Passo 2a: `int x = 5;` (dichiarazione interna — shadowing)
- Valuto: $\langle 5, \rho_1, \sigma_1 \rangle \Downarrow 5$
- Alloco **nuova** locazione: $l_2 \notin \text{dom}(\sigma_1)$
- $\rho_2 = \rho_1[x \mapsto l_2] = [x \mapsto l_2]$ (la nuova x **nasconde** quella esterna)
- $\sigma_2 = \sigma_1[l_2 \mapsto 5] = [l_1 \mapsto 10, l_2 \mapsto 5]$

> **Nota.** Ora esistono **due** locazioni: l_1 (la x esterna, valore 10) e l_2 (la x interna, valore 5). L'ambiente ρ_2 «vede» solo l_2 perché il binding $x \mapsto l_2$ ha sostituito $x \mapsto l_1$.

Passo 2b: `x := x + 1;` (assegnamento alla x interna)
- Valuto $x + 1$: $\langle x + 1, \rho_2, \sigma_2 \rangle$
  - $\rho_2(x) = l_2$, $\sigma_2(l_2) = 5$
  - $5 + 1 = 6$
- Aggiorno: $\sigma_3 = \sigma_2[l_2 \mapsto 6] = [l_1 \mapsto 10, l_2 \mapsto 6]$

Passo 3: Uscita dal blocco (regola Cmd-Block)
- L'ambiente torna a $\rho_1 = [x \mapsto l_1]$
- La memoria **rimane** $\sigma_3 = [l_1 \mapsto 10, l_2 \mapsto 6]$

> **Nota.** Dopo l'uscita dal blocco, la variabile x si riferisce di nuovo a l_1 (che contiene ancora 10). La locazione l_2 esiste ancora in memoria ma non è più accessibile tramite nessun nome.

Passo 4: `x := x * 2;` (assegnamento alla x esterna)
- Valuto $x * 2$: $\langle x * 2, \rho_1, \sigma_3 \rangle$
  - $\rho_1(x) = l_1$, $\sigma_3(l_1) = 10$
  - $10 \times 2 = 20$
- Aggiorno: $\sigma_4 = \sigma_3[l_1 \mapsto 20] = [l_1 \mapsto 20, l_2 \mapsto 6]$

Stato finale:
- $\rho_{\text{fin}} = [x \mapsto l_1]$
- $\sigma_{\text{fin}} = [l_1 \mapsto 20, l_2 \mapsto 6]$
- La variabile x (esterna) vale 20

Riepilogo dell'evoluzione degli ambienti:

Punto	**Ambiente ρ**	**x punta a**
Dopo `int x = 10;`	$[x \mapsto l_1]$	l_1 (valore 10)
Dentro blocco, dopo `int x = 5;`	$[x \mapsto l_2]$	l_2 (valore 5)
Dentro blocco, dopo `x := x + 1;`	$[x \mapsto l_2]$	l_2 (valore 6)
Dopo uscita dal blocco	$[x \mapsto l_1]$	l_1 (valore 10)
Dopo `x := x * 2;`	$[x \mapsto l_1]$	l_1 (valore 20)

Tabella 9: Evoluzione dell'ambiente con shadowing

3.8.1.4 Condizionali: calcolo del massimo

Esempio 3.27 – Sviluppo sequenziale: condizionali annidati (calcolo del massimo). Consideriamo il programma che trova il massimo tra tre numeri:

```
int a = 7;
int b = 3;
int c = 5;
int max = a;
if (b > max) { max := b; }
if (c > max) { max := c; }
```

Stato iniziale: $\rho_0 = \emptyset$, $\sigma_0 = \emptyset$

Passo 1–4: Dichiarazioni
- `int a = 7;`: alloco l_a, $\rho_1 = [a \mapsto l_a]$, $\sigma_1 = [l_a \mapsto 7]$
- `int b = 3;`: alloco l_b, $\rho_2 = [a \mapsto l_a, b \mapsto l_b]$, $\sigma_2 = [l_a \mapsto 7, l_b \mapsto 3]$
- `int c = 5;`: alloco l_c, $\rho_3 = [a \mapsto l_a, b \mapsto l_b, c \mapsto l_c]$, $\sigma_3 = [l_a \mapsto 7, l_b \mapsto 3, l_c \mapsto 5]$
- `int max = a;`: valuto a $(= 7)$, alloco l_m
  - $\rho_4 = [a \mapsto l_a, b \mapsto l_b, c \mapsto l_c, \max \mapsto l_m]$
  - $\sigma_4 = [l_a \mapsto 7, l_b \mapsto 3, l_c \mapsto 5, l_m \mapsto 7]$

Stato dopo le dichiarazioni:

$$\{a \mapsto 7, b \mapsto 3, c \mapsto 5, \max \mapsto 7\}$$

Passo 5: Primo condizionale `if (b > max) { max := b; }`
- Valuto la guardia: $\langle b > \max, \rho_4, \sigma_4 \rangle$
  - $\sigma_4(\rho_4(b)) = \sigma_4(l_b) = 3$
  - $\sigma_4(\rho_4(\max)) = \sigma_4(l_m) = 7$
  - $3 > 7 = \text{false}$
- Poiché la guardia è falsa, applichiamo (Cmd-IfFalse) con il ramo else implicito `skip;`
- Lo stato **non cambia**: $\sigma_5 = \sigma_4$

$$\{a \mapsto 7, b \mapsto 3, c \mapsto 5, \max \mapsto 7\} \text{ (invariato)}$$

Passo 6: Secondo condizionale `if (c > max) { max := c; }`
- Valuto la guardia: $\langle c > \max, \rho_4, \sigma_5 \rangle$
  - $\sigma_5(\rho_4(c)) = \sigma_5(l_c) = 5$
  - $\sigma_5(\rho_4(\max)) = \sigma_5(l_m) = 7$
  - $5 > 7 = \text{false}$
- Poiché la guardia è falsa, applichiamo (Cmd-IfFalse)
- Lo stato **non cambia**: $\sigma_6 = \sigma_5$

Stato finale:

$$\{a \mapsto 7, b \mapsto 3, c \mapsto 5, \max \mapsto 7\}$$

Il risultato è corretto: $\max = 7 = \max(7, 3, 5)$.

—

Variante: Consideriamo lo stesso programma con valori diversi: $a = 2$, $b = 8$, $c = 5$

Dopo le dichiarazioni: $\{a \mapsto 2, b \mapsto 8, c \mapsto 5, \max \mapsto 2\}$

Primo condizionale `if (b > max) { max := b; }`:
- Guardia: $8 > 2 = \text{true}$
- Eseguo `max := b;`: $\max = 8$
- Stato: $\{a \mapsto 2, b \mapsto 8, c \mapsto 5, \max \mapsto 8\}$

Secondo condizionale `if (c > max) { max := c; }`:
- Guardia: $5 > 8 = \text{false}$
- Stato invariato: $\{a \mapsto 2, b \mapsto 8, c \mapsto 5, \max \mapsto 8\}$

Risultato: $\max = 8 = \max(2, 8, 5)$.

—

Altra variante: $a = 2$, $b = 3$, $c = 9$

Dopo le dichiarazioni: $\{a \mapsto 2, b \mapsto 3, c \mapsto 9, \max \mapsto 2\}$

Primo condizionale:
- Guardia: $3 > 2 = \text{true}$
- Eseguo `max := b;`: $\max = 3$
- Stato: $\{a \mapsto 2, b \mapsto 3, c \mapsto 9, \max \mapsto 3\}$

Secondo condizionale:
- Guardia: $9 > 3 = \text{true}$
- Eseguo `max := c;`: $\max = 9$
- Stato: $\{a \mapsto 2, b \mapsto 3, c \mapsto 9, \max \mapsto 9\}$

Risultato: $\max = 9 = \max(2, 3, 9)$.

3.9 Terminazione e divergenza

Con l'introduzione dei cicli, i programmi possono non terminare mai. Questa distinzione tra terminazione e divergenza è fondamentale nello studio dei linguaggi di programmazione.

Definizione 3.23 – Terminazione. La **terminazione** è la proprietà di un programma che garantisce il completamento della sua esecuzione in tempo finito, producendo uno stato finale.

La terminazione di un ciclo `while(E){C}` richiede che:
- La guardia E contenga almeno una variabile.
- Il corpo C contenga almeno un assegnamento che modifichi quella variabile.
- Le modifiche successive portino la guardia a diventare falsa in un numero finito di passi.

Definizione 3.24 – Divergenza. La **divergenza** si verifica quando un programma (o un ciclo) non termina mai, continuando a eseguire istruzioni indefinitamente. In questo caso non si produce nessuno stato finale.

Esempio 3.28 – Programma divergente.

```
int x = 1;
while(x > 0){
    x := x + 1;
}
```

La guardia $x > 0$ è sempre vera (poiché x cresce ad ogni iterazione), quindi il ciclo non termina mai.

Nota. Non esiste un algoritmo che, dato un programma arbitrario, sia in grado di decidere se esso termina o meno. Questo risultato fondamentale è noto come **indecidibilità del problema della terminazione** (Halting Problem, Turing 1936).

3.9.1 Costrutti iterativi alternativi

In MAO è presente solamente il costrutto iterativo `while`, ma i linguaggi di programmazione comuni offrono diversi costrutti iterativi. Tutti questi costrutti possono essere espressi usando un ciclo `while`.

3.9.1.1 Ciclo for

Il ciclo `for` è un costrutto particolarmente compatto: in una sola riga è possibile leggere l'inizializzazione, la condizione di terminazione e l'aggiornamento.

Esempio 3.29 – for in JavaScript.

```
let somma = 0;
for (let i = 1; i < n; i = i + 1) {
    somma = somma + i;
}
```

Questo ciclo è equivalente al seguente codice MAO:

```
int somma = 0;
int i = 1;
while(i < n) {
    somma := somma + i;
    i := i + 1;
}
```

3.9.1.2 Ciclo do-while

Il ciclo `do-while` si utilizza quando il corpo deve essere eseguito almeno una volta, senza controllare preventivamente la condizione.

Esempio 3.30 – do-while in JavaScript.

```
let eta;
do {
    eta = parseInt(prompt("Anni?"));
} while(eta < 0);
```

In MAO, il `do-while` può essere simulato eseguendo il corpo una volta prima del `while`:

```
int eta = -1;
eta := /* leggi input */;
while(eta < 0) {
    eta := /* leggi input */;
}
```

4 Sistemi di Tipi, Funzioni e Ricorsione

4.1 Il linguaggio MAO e gli array

Il linguaggio MAO estende MiniMao con costrutti fondamentali per la programmazione reale: gli **array**, le **funzioni**, la **ricorsione** e un **sistema di tipi** formale. In questo capitolo si presenta ciascuno di questi aspetti, a partire dalla loro sintassi formale fino alla semantica operazionale e al type checking.

4.1.1 Array

> **Definizione 4.1 – Array – definizione informale.** Un **array** e una struttura dati che permette di trattare in modo efficiente un insieme finito di dati omogenei, cioe tutti dello stesso tipo. Gli elementi sono memorizzati in celle contigue di memoria e sono accessibili tramite un indice posizionale intero.

In MAO, gli array si dichiarano specificando il tipo degli elementi seguito da `[]`. Ad esempio, la dichiarazione

```
int[] voti = [18, 30, 23];
```

alloca in memoria un blocco contiguo di celle. La prima cella, situata all'*indirizzo base* l_b, contiene la lunghezza dell'array. Le celle successive $l_b + 1, l_b + 2, \ldots$ contengono i singoli elementi. Per accedere all'elemento in posizione i si usa la notazione $l_b[i]$, che corrisponde all'indirizzo $l_b + 1 + i$.

4.1.1.1 Rappresentazione in memoria

Nella struttura memoria-ambiente di MAO, un array viene rappresentato tramite una catena di indirezione. L'ambiente ρ associa il nome della variabile a una locazione l_x, e la memoria σ associa l_x all'indirizzo base l_b dell'array:

$$\begin{gathered} \rho(\text{voti}) = l_x, \quad \sigma(l_x) = l_b \\ \sigma(l_b) = 3 \qquad \textit{(lunghezza)} \\ \sigma(l_b + 1) = 18, \quad \sigma(l_b + 2) = 30, \quad \sigma(l_b + 3) = 23 \end{gathered}$$

Si osservi che la variabile `voti` non contiene direttamente i dati dell'array, ma un *riferimento* (locazione) all'indirizzo base. Questa scelta progettuale ha conseguenze importanti, come vedremo nella sezione sull'aliasing.

> **Definizione 4.2 – Array – definizione formale.** Un array e una collezione finita di elementi dello stesso tipo, memorizzati in celle contigue di memoria. Il numero degli elementi e detto lunghezza dell'array. Tipo e lunghezza sono fissati al momento della dichiarazione e sono **statici**: non possono essere modificati durante l'esecuzione del programma.

Un array permette di trattare come entita atomiche intere collezioni di dati e, al contempo, consente l'accesso ai singoli elementi tramite indici posizionali. Gli indici ammissibili appartengono all'intervallo

$$[0, \text{lunghezza})$$

cioe da 0 (incluso) a «lunghezza» -1 (incluso). Un accesso con indice fuori da questo intervallo costituisce un errore a tempo di esecuzione.

4.1.2 Sintassi degli array

Per ottenere la lunghezza di un array si utilizza il costrutto `.length`, che restituisce un valore intero.

> **Esempio 4.1 – Lunghezza di un array.**
>
> ```
> int[] voti = [18, 30, 23];
>
> voti.length = 3
> ```

La grammatica di MAO viene estesa con le seguenti produzioni per supportare gli array:

$$T ::= \ldots \mid T[]$$
$$E ::= \ldots \mid [\ S\ ] \mid \text{new}\ \ T\ [\ E\ ] \mid E.\ \text{length} \mid E\ [\ E\ ]$$
$$S ::= E \mid E\ ,\ S$$
$$C ::= \ldots \mid E\ [\ E\ ] := E$$

Dove ciascuna produzione ha il seguente significato:

- $T[]$: tipo array di elementi di tipo T (ad esempio `int[]`, `bool[]`)
- $[\ S\ ]$: **letterale array**, cioe una lista esplicita di espressioni che definisce i valori iniziali
- new $T\ [\ E\ ]$: **allocazione** di un nuovo array di tipo T con dimensione data dalla valutazione di E, con elementi inizializzati ai valori di default
- $E.$ length: espressione che restituisce la **lunghezza** dell'array
- $E\ [\ E\ ]$: **accesso** a un singolo elemento dell'array, dove la prima espressione denota l'array e la seconda l'indice
- $E\ [\ E\ ] := E$: **assegnamento** a un elemento dell'array

4.1.3 Valori e riferimenti

In MiniMao le celle di memoria contenevano solamente valori interi e booleani. L'ambiente e la memoria erano definiti come:

$$\rho : \mathbb{I} \hookrightarrow \mathbb{L} \qquad \sigma : \mathbb{L} \hookrightarrow \mathbb{V} \text{ dove } \mathbb{V} = \mathbb{Z} \cup \mathbb{B}$$

Con l'introduzione degli array in MAO, il dominio dei valori viene esteso per includere anche le *locazioni* (indirizzi di memoria), poiche una variabile di tipo array contiene un riferimento all'indirizzo base dell'array:

$$\rho : \mathbb{I} \hookrightarrow \mathbb{L} \qquad \sigma : \mathbb{L} \hookrightarrow \mathbb{V} \text{ dove } \mathbb{V} = \mathbb{Z} \cup \mathbb{B} \cup \mathbb{L}$$

Le locazioni che compaiono come valori prendono il nome di **riferimenti**. Un riferimento non e un dato del programma, ma un indirizzo di memoria che permette di accedere indirettamente a una struttura dati.

4.1.4 Espressioni pure e con effetti collaterali

> **Definizione 4.3 – Espressione pura.** Un'espressione si dice **pura** se la sua valutazione non modifica lo stato della memoria. Il risultato dipende solo dai valori correnti delle variabili, senza effetti collaterali.

In **MiniMao** tutte le espressioni erano pure: la valutazione di un'espressione E produceva un valore v lasciando la memoria σ invariata. Cio permetteva di usare la notazione semplificata:

$$\langle E, \rho, \sigma \rangle \Downarrow v$$

Esempio 4.2 – Espressioni pure in MiniMao. Le seguenti espressioni non modificano la memoria:
- `x + 3` → legge il valore di x, calcola la somma, restituisce un intero
- `a > b and c` → legge a, b, c, restituisce un booleano
- `(x + 1) * (y - 2)` → solo letture e calcoli aritmetici

In **MAO**, con l'introduzione degli array, alcune espressioni possono **allocare nuova memoria**. Queste espressioni producono **effetti collaterali** e restituiscono, oltre al valore, una memoria modificata σ':

$$\langle E, \rho, \sigma \rangle \Downarrow (v, \sigma')$$

Esempio 4.3 – Espressioni con effetti collaterali. Le seguenti espressioni modificano la memoria allocando nuove celle:
- `new int[5]` → alloca 6 celle consecutive (1 per la lunghezza + 5 per gli elementi)
- `[1, 2, 3]` → alloca 4 celle consecutive (1 per la lunghezza + 3 per i valori)
- `new bool[n+1]` → prima valuta l'espressione `n+1`, poi alloca il numero risultante di celle

Nota – Conseguenza sulla semantica. In MAO la valutazione di *qualsiasi* espressione restituisce sempre una coppia (v, σ'), anche quando $\sigma' = \sigma$ (cioe quando l'espressione e pura). Questo garantisce uniformita nella semantica: ogni regola di valutazione segue lo stesso schema, indipendentemente dalla presenza o meno di effetti collaterali.

La distinzione tra espressioni pure e non pure ha importanti implicazioni pratiche:
- **Ordine di valutazione**: se due sottoespressioni hanno effetti collaterali, l'ordine in cui vengono valutate puo influenzare il risultato finale
- **Ottimizzazioni del compilatore**: il compilatore puo riordinare o eliminare espressioni pure senza alterare la semantica del programma, ma non puo fare altrettanto con espressioni che hanno effetti collaterali
- **Ragionamento formale**: le espressioni pure sono piu semplici da analizzare perche il loro comportamento e completamente determinato dai valori correnti delle variabili

4.1.5 Semantica operazionale degli array

Le regole seguenti definiscono la semantica operazionale degli array in stile *big-step*. In ciascuna regola, le premesse (sopra la linea) stabiliscono le condizioni necessarie, e la conclusione (sotto la linea) descrive il risultato della valutazione.

4.1.5.1 Allocazione di un array letterale

Quando si valuta un letterale array $[E_1, ..., E_n]$, si valutano in ordine le n espressioni, ottenendo i valori $v_1, ..., v_n$. Si allocano poi $n + 1$ celle consecutive a partire dall'indirizzo base l_b: la prima cella memorizza la lunghezza n, le successive i valori degli elementi.

$$\frac{\langle E_1, \rho, \sigma \rangle \Downarrow (v_1, \sigma_1) \quad ... \quad \langle E_n, \rho, \sigma_{n-1} \rangle \Downarrow (v_n, \sigma_n) \quad l_b, l_b + 1, ..., l_b + n \notin \mathrm{dom}(\sigma_n)}{\langle [E_1, ..., E_n], \rho, \sigma \rangle \Downarrow \left(l_b, \sigma_{n[l_b \mapsto n]}[l_b + 1 \mapsto v_1]...[l_b + n \mapsto v_n] \right)} \quad \text{(Array-Lit)}$$

Nota. Si osservi che il valore restituito e l'indirizzo base l_b, non l'array stesso. Cio riflette il fatto che in MAO gli array sono gestiti per riferimento. Inoltre, ogni espressione E_i viene valutata nella memoria σ_{i-1} risultante dalla valutazione precedente, garantendo la corretta propagazione degli effetti collaterali.

4.1.5.2 Allocazione con new

L'espressione `new T[E]` valuta prima E per ottenere la dimensione n, poi alloca $n+1$ celle consecutive: la prima per la lunghezza, le restanti inizializzate al valore di default del tipo T (tipicamente 0 per `int`, `false` per `bool`).

$$\frac{\langle E, \rho, \sigma\rangle \Downarrow (n, \sigma') \quad l_b, ..., l_b + n \notin \mathrm{dom}(\sigma')}{\langle \text{new } T[E], \rho, \sigma\rangle \Downarrow (l_b, \sigma'[l_b \mapsto n][l_b + 1 \mapsto \text{default}]...[l_b + n \mapsto \text{default}])} \quad \text{(Array-New)}$$

4.1.5.3 Lunghezza

L'espressione `E.length` valuta E ottenendo l'indirizzo base l_b, quindi legge il valore memorizzato in l_b, che per costruzione e la lunghezza dell'array.

$$\frac{\langle E, \rho, \sigma\rangle \Downarrow (l_b, \sigma') \quad \sigma'(l_b) = n}{\langle E.\text{length}, \rho, \sigma\rangle \Downarrow (n, \sigma')} \quad \text{(Array-Length)}$$

4.1.5.4 Accesso a un elemento

L'espressione $E_1[E_2]$ valuta prima E_1 per ottenere l'indirizzo base l_b, poi E_2 per ottenere l'indice i. La premessa $0 \leq i < \sigma_2(l_b)$ garantisce che l'indice sia nell'intervallo valido. L'elemento cercato si trova all'indirizzo $l_b + 1 + i$ (si aggiunge 1 perche la prima cella contiene la lunghezza).

$$\frac{\langle E_1, \rho, \sigma\rangle \Downarrow (l_b, \sigma_1) \quad \langle E_2, \rho, \sigma_1\rangle \Downarrow (i, \sigma_2) \quad 0 \leq i < \sigma_2(l_b)}{\langle E_1[E_2], \rho, \sigma\rangle \Downarrow (\sigma_2(l_b + 1 + i), \sigma_2)} \quad \text{(Array-Access)}$$

4.1.5.5 Assegnamento in array

Il comando $E_1[E_2] := E_3$ valuta le tre espressioni in ordine: E_1 per l'indirizzo base, E_2 per l'indice, E_3 per il nuovo valore. La cella all'indirizzo $l_b + 1 + i$ viene aggiornata con il valore v.

$$\frac{\langle E_1, \rho, \sigma\rangle \Downarrow (l_b, \sigma_1) \quad \langle E_2, \rho, \sigma_1\rangle \Downarrow (i, \sigma_2) \quad \langle E_3, \rho, \sigma_2\rangle \Downarrow (v, \sigma_3)}{\langle E_1[E_2] := E_3;, \rho, \sigma\rangle \to \langle \rho, \sigma_3[l_b + 1 + i \mapsto v]\rangle} \quad \text{(Array-Assign)}$$

4.1.6 Aliasing

Definizione 4.4 – Aliasing. Si ha **aliasing** quando due o piu variabili distinte fanno riferimento alla stessa zona di memoria. Poiche in MAO gli array sono gestiti tramite riferimenti (indirizzi base), l'assegnamento di un array a un'altra variabile copia il *riferimento*, non i dati. Di conseguenza, entrambe le variabili puntano allo stesso blocco di memoria.

Esempio 4.4 – Problema dell'aliasing. Consideriamo il seguente frammento di codice:

```
int[] a = [1, 2, 3];
int[] b = a;        // b punta allo stesso array di a!
b[0] := 99;         // modifica anche a[0]!
```

Dopo l'esecuzione, lo stato e il seguente:

- $\rho = [a \mapsto l_a, b \mapsto l_b]$
- $\sigma(l_a) = \sigma(l_b) = l_{\text{base}}$ (stesso indirizzo base!)

Quindi `a[0]` e `b[0]` accedono alla stessa cella di memoria $l_{\text{base}} + 1$. La modifica effettuata tramite `b` e visibile anche tramite `a`, perche non e stata creata una copia indipendente dell'array.

> **Nota – Conseguenza sul passaggio di parametri.** L'aliasing ha implicazioni dirette sul passaggio di parametri alle funzioni. Quando si passa un array come argomento a una funzione, si passa il suo **riferimento** (indirizzo base). Di conseguenza, eventuali modifiche agli elementi dell'array effettuate nel corpo della funzione si riflettono sull'array originale del chiamante. Questo comportamento e equivalente a un passaggio per *riferimento implicito*.

4.2 Analisi statica e sistema di tipi

In un linguaggio di programmazione, le grammatiche definiscono in modo rigoroso le categorie sintattiche delle espressioni e dei comandi. Tuttavia, molti programmi sintatticamente validi possono contenere errori che si manifestano solo a tempo di esecuzione: ad esempio, sommare un intero con un booleano, oppure accedere a un array con un indice di tipo booleano. Per prevenire questa classe di errori si ricorre all'**analisi statica**.

> **Definizione 4.5 – Analisi statica.** L'**analisi statica** consiste nei controlli effettuati sul codice sorgente *senza eseguire il programma*. Il suo scopo e individuare errori e anomalie prima dell'esecuzione, basandosi esclusivamente sulla struttura sintattica e sulle informazioni di tipo.

La maggior parte dei linguaggi di programmazione moderni prevede diversi controlli di analisi statica (ad esempio, il controllo di raggiungibilita del codice, l'analisi di variabili non inizializzate, e altri). In MAO ci si limita al controllo dei tipi (type checking), che verifica la coerenza tra i tipi dichiarati per le variabili e l'uso che ne viene fatto nelle espressioni e nei comandi.

4.2.1 Sistemi di tipi

In MAO il controllo dei tipi avviene in modo formale attraverso regole di tipo che stabiliscono le condizioni necessarie affinche un'espressione o un comando vengano considerati ben tipati. Il controllo e **composizionale**: le regole di tipo per un costrutto composito sono definite in funzione dei giudizi di tipo sulle sue sottocomponenti. Per poter effettuare questo controllo e necessario conoscere i tipi associati alle variabili che occorrono nelle espressioni e nei comandi; a tal fine si introduce l'*ambiente di tipo*.

4.2.2 Ambiente di tipo

> **Definizione 4.6 – Ambiente di tipo.** L'**ambiente di tipo** $\Gamma : \mathbb{I} \hookrightarrow \mathbb{T}$ e una funzione parziale che associa a ciascun identificatore nel suo dominio un tipo. La scrittura $\Gamma(x) = \text{int}$ si legge: «nell'ambiente Γ si assume che la variabile x abbia tipo `int`».

Per comodita si scrive $\text{Id} : T$ in luogo di $\Gamma(\text{Id}) = T$. Un ambiente di tipo si rappresenta come un insieme di associazioni:

$$\Gamma = \{\text{Id}_1 : T_1, \text{Id}_2 : T_2, ..., \text{Id}_n : T_n\}$$

L'ambiente di tipo e un concetto esclusivamente statico: esiste solo a tempo di compilazione e serve al type checker per verificare la correttezza del programma. Non va confuso con l'ambiente ρ (che mappa identificatori a locazioni a tempo di esecuzione).

> **Esempio 4.5 – Costruzione di un ambiente di tipo.** Dopo le seguenti dichiarazioni:
>
> ```
> int temp = 2;
> bool y = true;
> ```

L'ambiente di tipo risultante e $\Gamma = \{\text{temp} : \text{int}, y : \text{bool}\}$.

4.2.3 Variabili libere

Le variabili libere di un'espressione o di un comando sono quelle variabili che vi occorrono senza essere state dichiarate localmente. La loro definizione formale e necessaria per garantire che il type checking possa procedere: ogni variabile libera deve essere presente nell'ambiente di tipo Γ.

4.2.3.1 Variabili libere in un'espressione

Data un'espressione $E \in \mathbb{E}$, la funzione $\text{fv}(.) : \mathbb{E} \to \mathcal{P}_{\text{fin}}(\mathbb{I})$ restituisce l'insieme finito di tutte le variabili che occorrono in E, dette variabili libere. La funzione e definita per induzione sulla struttura dell'espressione:

$$\begin{gathered}
\text{fv}(n) = \emptyset \\
\text{fv}(\text{true}) = \emptyset \\
\text{fv}(\text{false}) = \emptyset \\
\text{fv}(\text{Id}) = \{\text{Id}\} \\
\text{fv}(E_1 \text{ bop } E_2) = \text{fv}(E_1) \cup \text{fv}(E_2) \\
\text{fv}(\text{uop } E) = \text{fv}(E) \\
\text{fv}((E)) = \text{fv}(E) \\
\text{fv}(E_1[E_2]) = \text{fv}(E_1) \cup \text{fv}(E_2) \\
\text{fv}(E.\text{length}) = \text{fv}(E) \\
\text{fv}(\text{new } T[E]) = \text{fv}(E)
\end{gathered}$$

Le costanti (numeriche e booleane) non contengono variabili libere. Un identificatore ha se stesso come unica variabile libera. Per le espressioni composte, le variabili libere sono l'unione delle variabili libere delle sottoespressioni.

Esempio 4.6. $\text{fv}(\texttt{x+3}) = \text{fv}(\texttt{x}) \cup \text{fv}(\texttt{3}) = \{\texttt{x}\} \cup \emptyset = \{\texttt{x}\}$

Esempio 4.7. $\text{fv}(\texttt{new int[a.length]}) = \text{fv}(\texttt{a.length}) = \text{fv}(\texttt{a}) = \{\texttt{a}\}$

Esempio 4.8. $\text{fv}(\texttt{x\%7 == z*x}) = \text{fv}(\texttt{x\%7}) \cup \text{fv}(\texttt{z*x}) = \{\texttt{x}\} \cup \{\texttt{z, x}\} = \{\texttt{x, z}\}$

4.2.3.2 Variabili libere in un comando

Dato un comando $C \in \mathbb{C}$, la funzione $\text{fv}(.) : \mathbb{C} \to \mathcal{P}_{\text{fin}}(\mathbb{I})$ restituisce l'insieme di tutte le variabili che occorrono in C senza essere state dichiarate all'interno di C stesso. La definizione per induzione richiede una funzione ausiliaria $\text{dv}(.) : \mathbb{C} \to \mathcal{P}_{\text{fin}}(\mathbb{I})$, che restituisce l'insieme delle variabili *introdotte* da dichiarazioni.

$$\begin{gathered}
\text{fv}(\text{skip};) = \emptyset \\
\text{fv}(T \;\; \text{Id} = E;) = \text{fv}(E) \\
\text{fv}(\text{Id} := E;) = \{\text{Id}\} \cup \text{fv}(E) \\
\text{fv}(E_1[E_2] := E_3;) = \text{fv}(E_1) \cup \text{fv}(E_2) \cup \text{fv}(E_3) \\
\text{fv}(C_1 C_2) = \text{fv}(C_1) \cup (\text{fv}(C_2) \setminus \text{dv}(C_1)) \\
\text{fv}(\{C\}) = \text{fv}(C) \\
\text{fv}(\text{if}(E)\{C_1\}\text{else}\{C_2\}) = \text{fv}(E) \cup \text{fv}(C_1) \cup \text{fv}(C_2) \\
\text{fv}(\text{while}(E)\{C\}) = \text{fv}(E) \cup \text{fv}(C)
\end{gathered}$$

> **Nota.** Nella regola per la sequenza C_1C_2, le variabili dichiarate in C_1 (cioe $\mathrm{dv}(C_1)$) vengono sottratte dalle variabili libere di C_2, perche le dichiarazioni di C_1 rendono disponibili quei nomi nel contesto di C_2.

La funzione $\mathrm{dv}(C)$ e definita come segue:

$$\mathrm{dv}(T\ \ \mathrm{Id} = E;) = \{\mathrm{Id}\}$$
$$\mathrm{dv}(C_1C_2) = \mathrm{dv}(C_1) \cup \mathrm{dv}(C_2)$$
$$\mathrm{dv}(\text{altri comandi}) = \emptyset$$

4.2.4 Giudizi di tipo

I giudizi di tipo sono le asserzioni fondamentali del sistema di tipi. Ve ne sono di due forme, una per le espressioni e una per i comandi.

4.2.4.1 Giudizi di tipo per le espressioni

Dato un ambiente di tipo Γ e un'espressione E tale che $\mathrm{fv}(E) \subseteq \mathrm{dom}(\Gamma)$, possiamo derivare un giudizio di tipo della forma

$$\Gamma \vdash E : T$$

che si legge: «nell'ambiente Γ, l'espressione E e ben tipata e ha tipo T». La condizione $\mathrm{fv}(E) \subseteq \mathrm{dom}(\Gamma)$ garantisce che tutte le variabili che compaiono in E abbiano un tipo noto.

4.2.4.2 Giudizi di tipo per i comandi

Dato un ambiente di tipo Γ e un comando C tale che $\mathrm{fv}(C) \subseteq \mathrm{dom}(\Gamma)$, possiamo derivare un giudizio della forma

$$\Gamma \vdash C : \Gamma'$$

che si legge: «nell'ambiente Γ, il comando C e ben tipato e produce l'ambiente locale Γ'». L'ambiente Γ' contiene le associazioni introdotte dalle eventuali dichiarazioni presenti in C; per i comandi che non dichiarano variabili (come l'assegnamento, il condizionale o il ciclo), si ha $\Gamma' = \emptyset$.

4.2.5 Regole di type checking per le espressioni

Le regole seguenti definiscono come derivare i giudizi di tipo per le espressioni. Ogni regola e presentata in stile *regola di inferenza*: le premesse si trovano sopra la linea orizzontale, la conclusione sotto.

4.2.5.1 Costanti

Una costante intera n ha sempre tipo `int`, indipendentemente dall'ambiente. Le costanti booleane `true` e `false` hanno tipo `bool`. Queste sono regole *assiomatiche* (senza premesse sostanziali).

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \mathrm{int}} \quad \text{(T-Int)}$$

$$\frac{}{\Gamma \vdash \mathrm{true} : \mathrm{bool}} \quad \text{(T-True)} \qquad \frac{}{\Gamma \vdash \mathrm{false} : \mathrm{bool}} \quad \text{(T-False)}$$

4.2.5.2 Variabili

Il tipo di un identificatore e quello assegnato dall'ambiente Γ. La premessa richiede che l'identificatore sia presente nel dominio di Γ.

$$\frac{\Gamma(\text{Id}) = T}{\Gamma \vdash \text{Id} : T} \quad \text{(T-Var)}$$

4.2.5.3 Operatori aritmetici

Gli operatori aritmetici $(+, -, \times, \div, \text{mod})$ richiedono che entrambi gli operandi abbiano tipo `int` e producono un risultato di tipo `int`.

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 \text{ aop } E_2 : \text{int}} \quad \text{(T-Aop) dove aop} \in \{+, -, \times, \div, \text{mod}\}$$

4.2.5.4 Operatori di confronto

Gli operatori di confronto di ordinamento $(<, \leq, >, \geq)$ confrontano due espressioni intere e producono un risultato booleano:

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 \text{ cop } E_2 : \text{bool}} \quad \text{(T-Cop) dove cop} \in \{<, \leq, >, \geq\}$$

Gli operatori di uguaglianza e disuguaglianza $(==, \neq)$ accettano operandi dello stesso tipo, sia `int` che `bool`, e producono un risultato booleano. Si richiede che entrambi gli operandi abbiano lo stesso tipo $T \in \{\text{int}, \text{bool}\}$:

$$\frac{\Gamma \vdash E_1 : T \quad \Gamma \vdash E_2 : T \quad T \in \{\text{int}, \text{bool}\}}{\Gamma \vdash E_1 \text{ eop } E_2 : \text{bool}} \quad \text{(T-Eop) dove eop} \in \{==, \neq\}$$

> **Nota.** La distinzione tra (T-Cop) e (T-Eop) riflette il fatto che gli operatori di ordinamento hanno senso solo su valori interi (non ha significato confrontare `true < false`), mentre l'uguaglianza e la disuguaglianza sono definite anche per i booleani. Ad esempio, l'espressione `(x > 0) == (y > 0)` e ben tipata: entrambi i lati hanno tipo `bool` e l'operatore `==` accetta operandi booleani.

4.2.5.5 Operatori logici

Gli operatori logici binari ($\wedge$, $\vee$) richiedono operandi booleani e producono un booleano. L'operatore unario $\neg$ nega un'espressione booleana.

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : \text{bool}}{\Gamma \vdash E_1 \text{ lop } E_2 : \text{bool}} \quad \text{(T-Lop) dove lop} \in \{\wedge, \vee\}$$

$$\frac{\Gamma \vdash E : \text{bool}}{\Gamma \vdash \neg E : \text{bool}} \quad \text{(T-Not)}$$

4.2.5.6 Regole di tipo per gli array

Le seguenti regole gestiscono il type checking dei costrutti relativi agli array.

$$\frac{\Gamma \vdash E : \text{int}}{\Gamma \vdash \text{new } T[E] : T[]} \quad \text{(T-NewArray)}$$

Un letterale array $[E_1, ..., E_n]$ e ben tipato se tutte le espressioni hanno lo stesso tipo T. Il risultato ha tipo $T[]$:

$$\frac{\Gamma \vdash E_1 : T \quad ... \quad \Gamma \vdash E_n : T}{\Gamma \vdash [E_1, ..., E_n] : T[]} \qquad \text{(T-ArrayLit)}$$

> **Nota.** La regola (T-ArrayLit) garantisce l'omogeneita del tipo degli elementi: un letterale come `[1, 2, 3]` ha tipo `int[]`, mentre `[true, false]` ha tipo `bool[]`. Un letterale misto come `[1, true, 3]` e *mal tipato* perche non esiste un tipo T tale che tutte le espressioni abbiano tipo T.

$$\frac{\Gamma \vdash E : T[]}{\Gamma \vdash E.\text{length} : \text{int}} \qquad \text{(T-Length)}$$

$$\frac{\Gamma \vdash E_1 : T[] \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1[E_2] : T} \qquad \text{(T-ArrayAccess)}$$

4.2.6 Regole di type checking per i comandi

Le regole per i comandi stabiliscono quando un comando e ben tipato e quale ambiente locale Γ' esso produce.

4.2.6.1 Skip

Il comando `skip` e sempre ben tipato e non introduce nuove variabili.

$$\frac{}{\Gamma \vdash \text{skip}; : \emptyset} \qquad \text{(T-Skip)}$$

4.2.6.2 Dichiarazione

Una dichiarazione `T Id = E;` e ben tipata se l'espressione E ha tipo T. La dichiarazione introduce l'associazione $\text{Id} : T$ nell'ambiente locale.

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash T \;\; \text{Id} = E; : \{\text{Id} : T\}} \qquad \text{(T-Decl)}$$

4.2.6.3 Assegnamento

Un assegnamento `Id := E;` e ben tipato se la variabile `Id` e gia presente nell'ambiente con tipo T e l'espressione E ha lo stesso tipo T. L'assegnamento non introduce nuove variabili.

$$\frac{\Gamma(\text{Id}) = T \quad \Gamma \vdash E : T}{\Gamma \vdash \text{Id} := E; : \emptyset} \qquad \text{(T-Assign)}$$

4.2.6.4 Assegnamento in array

L'assegnamento a un elemento di array richiede che E_1 abbia tipo $T[]$, che E_2 abbia tipo `int` (l'indice) e che E_3 abbia tipo T (coerente con il tipo degli elementi).

$$\frac{\Gamma \vdash E_1 : T[] \quad \Gamma \vdash E_2 : \text{int} \quad \Gamma \vdash E_3 : T}{\Gamma \vdash E_1[E_2] := E_3; : \emptyset} \qquad \text{(T-ArrayAssign)}$$

4.2.6.5 Sequenza

La composizione sequenziale $C_1 C_2$ verifica prima C_1 nell'ambiente Γ, ottenendo l'ambiente locale Γ_1. Poi verifica C_2 nell'ambiente esteso $\Gamma \cup \Gamma_1$, ottenendo Γ_2. L'ambiente locale complessivo e l'unione $\Gamma_1 \cup \Gamma_2$.

$$\frac{\Gamma \vdash C_1 : \Gamma_1 \quad \Gamma \cup \Gamma_1 \vdash C_2 : \Gamma_2}{\Gamma \vdash C_1 C_2 : \Gamma_1 \cup \Gamma_2} \quad \text{(T-Seq)}$$

4.2.6.6 Blocco

Un blocco `{C}` incapsula un comando: le dichiarazioni all'interno del blocco sono *locali* e non visibili all'esterno. Per questo motivo la regola restituisce l'ambiente vuoto $\emptyset$.

$$\frac{\Gamma \vdash C : \Gamma'}{\Gamma \vdash \{C\} : \emptyset} \quad \text{(T-Block)}$$

4.2.6.7 Condizionale

Il condizionale richiede che la guardia E sia di tipo `bool` e che entrambi i rami C_1 e C_2 siano ben tipati. Non introduce nuove variabili nell'ambiente esterno, perche i rami sono racchiusi in blocchi.

$$\frac{\Gamma \vdash E : \text{bool} \quad \Gamma \vdash C_1 : \Gamma_1 \quad \Gamma \vdash C_2 : \Gamma_2}{\Gamma \vdash \text{if}(E)\{C_1\}\text{else}\{C_2\} : \emptyset} \quad \text{(T-If)}$$

4.2.6.8 Ciclo while

Il ciclo `while` richiede che la guardia E abbia tipo `bool` e che il corpo C sia ben tipato. Come il condizionale, non introduce nuove variabili nell'ambiente esterno.

$$\frac{\Gamma \vdash E : \text{bool} \quad \Gamma \vdash C : \Gamma'}{\Gamma \vdash \text{while}(E)\{C\} : \emptyset} \quad \text{(T-While)}$$

4.2.7 Esempi di derivazioni di tipo

I seguenti esempi illustrano come le regole di type checking vengano applicate per costruire *alberi di derivazione* che dimostrano la correttezza dei tipi in un programma.

> **Esempio 4.9 – Derivazione di tipo per un'espressione aritmetica.** Verifichiamo che l'espressione `x + y * 2` sia ben tipata nell'ambiente $\Gamma = \{x : \text{int}, y : \text{int}\}$.
>
> L'albero di derivazione si costruisce dal basso verso l'alto, partendo dalle foglie (assiomi) e applicando le regole fino a raggiungere il giudizio desiderato.
>
> Prima si verifica la sottoespressione `y * 2`:
>
> $$\frac{\dfrac{\Gamma(y) = \text{int}}{\Gamma \vdash y : \text{int}}\ \text{(T-Var)} \qquad \dfrac{2 \in \mathbb{Z}}{\Gamma \vdash 2 : \text{int}}\ \text{(T-Int)}}{\Gamma \vdash y * 2 : \text{int}} \quad \text{(T-Aop)}$$
>
> Poi si combina con `x`:
>
> $$\frac{\Gamma(x) = \text{int} \qquad \Gamma \vdash y * 2 : \text{int}}{\Gamma \vdash x + y * 2 : \text{int}} \quad \text{(T-Aop)}$$
>
> L'espressione e **ben tipata** con tipo `int`.

> **Esempio 4.10 – Derivazione di tipo per una sequenza di comandi.** Verifichiamo il seguente frammento:
>
> ```
> int z = 0;
> z := x + 1;
> ```

nell'ambiente iniziale $\Gamma_0 = \{x : \text{int}\}$.

Passo 1: Type checking della dichiarazione `int z = 0;`

$$\frac{\Gamma_0 \vdash 0 : \text{int}}{\Gamma_0 \vdash \text{int}\ \ z = 0; : \{z : \text{int}\}} \quad \text{(T-Decl)}$$

Dopo la dichiarazione, l'ambiente viene esteso: $\Gamma_1 = \Gamma_0 \cup \{z : \text{int}\} = \{x : \text{int}, z : \text{int}\}$

Passo 2: Type checking dell'assegnamento `z := x + 1;` nell'ambiente Γ_1

Prima verifichiamo l'espressione `x + 1`:

$$\frac{\Gamma_1 \vdash x : \text{int} \quad \Gamma_1 \vdash 1 : \text{int}}{\Gamma_1 \vdash x + 1 : \text{int}} \quad \text{(T-Aop)}$$

Poi l'assegnamento, verificando la coerenza dei tipi:

$$\frac{\Gamma_1(z) = \text{int} \quad \Gamma_1 \vdash x + 1 : \text{int}}{\Gamma_1 \vdash z := x + 1; : \emptyset} \quad \text{(T-Assign)}$$

Il comando e **ben tipato**.

Esempio 4.11 – Errore di tipo. Consideriamo il seguente programma:

```
int x = 5;
bool y = true;
x := x + y;  // ERRORE!
```

Con $\Gamma = \{x : \text{int}, y : \text{bool}\}$, tentiamo di derivare il tipo di `x + y`.

- $\Gamma \vdash x : \text{int}$ (corretto per T-Var)
- $\Gamma \vdash y : \text{bool}$ (corretto per T-Var)
- La regola (T-Aop) richiede che **entrambi** gli operandi siano di tipo `int`:

$$\frac{\Gamma \vdash E_1 : \text{int} \qquad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 + E_2 : \text{int}}$$

Poiche y ha tipo `bool` e non `int`, la premessa $\Gamma \vdash y : \text{int}$ non e derivabile. La derivazione **fallisce** e il compilatore segnala un errore di tipo. Il programma e **mal tipato**.

Esempio 4.12 – Type checking di un programma con while. Verifichiamo il type checking del seguente programma completo:

```
int x = 5;
int y = 6;
while (x != y) {
  if (x < y) { x := x + 2; }
  else { y := y + 1; }
}
```

Passo 1: Partiamo dall'ambiente vuoto $\Gamma_0 = \emptyset$.

Passo 2: Type checking di `int x = 5;`

$$\frac{5 \in \mathbb{Z}}{\dfrac{\Gamma_0 \vdash 5 : \text{int}}{\Gamma_0 \vdash \text{int}\ \ x = 5; : \{x : \text{int}\}}} \quad \text{(T-Decl)}$$

Dopo la dichiarazione: $\Gamma_1 = \Gamma_0 \cup \{x : \text{int}\} = \{x : \text{int}\}$

Passo 3: Type checking di `int y = 6;` nell'ambiente Γ_1

$$\frac{6 \in \mathbb{Z}}{\dfrac{\Gamma_1 \vdash 6 : \text{int}}{\Gamma_1 \vdash \text{int}\ \ y = 6; : \{y : \text{int}\}}}\ \text{(T-Decl)}$$

Dopo la dichiarazione: $\Gamma_2 = \Gamma_1 \cup \{y : \text{int}\} = \{x : \text{int}, y : \text{int}\}$

Passo 4: Verifica della guardia `x != y` nell'ambiente Γ_2

$$\frac{\dfrac{\Gamma_2(x) = \text{int} \qquad \Gamma_2(y) = \text{int}}{\Gamma_2 \vdash x : \text{int} \qquad \Gamma_2 \vdash y : \text{int}}}{\Gamma_2 \vdash x \neq y : \text{bool}}\ \text{(T-Cop)}$$

Passo 5: Verifica del ramo then `{ x := x + 2; }`

Prima l'espressione `x + 2`:

$$\frac{\dfrac{\Gamma_2(x) = \text{int} \qquad 2 \in \mathbb{Z}}{\Gamma_2 \vdash x : \text{int} \qquad \Gamma_2 \vdash 2 : \text{int}}}{\Gamma_2 \vdash x + 2 : \text{int}}\ \text{(T-Aop)}$$

Poi l'assegnamento e il blocco:

$$\frac{\dfrac{\Gamma_2(x) = \text{int} \quad \Gamma_2 \vdash x + 2 : \text{int}}{\Gamma_2 \vdash x := x + 2; : \emptyset}\ \text{(T-Assign)}}{\Gamma_2 \vdash \{x := x + 2;\} : \emptyset}\ \text{(T-Block)}$$

Passo 6: Verifica del ramo else `{ y := y + 1; }` (procedimento analogo)

$$\frac{\dfrac{\Gamma_2(y) = \text{int} \quad \Gamma_2 \vdash y + 1 : \text{int}}{\Gamma_2 \vdash y := y + 1; : \emptyset}\ \text{(T-Assign)}}{\Gamma_2 \vdash \{y := y + 1;\} : \emptyset}\ \text{(T-Block)}$$

Passo 7: Verifica della guardia `x < y`:

$$\frac{\Gamma_2 \vdash x : \text{int} \quad \Gamma_2 \vdash y : \text{int}}{\Gamma_2 \vdash x < y : \text{bool}}\ \text{(T-Cop)}$$

Passo 8: Verifica del condizionale completo:

$$\frac{\Gamma_2 \vdash x < y : \text{bool} \quad \Gamma_2 \vdash \{x := x + 2;\} : \emptyset \quad \Gamma_2 \vdash \{y := y + 1;\} : \emptyset}{\Gamma_2 \vdash \text{if}(x < y)\{x := x + 2;\}\text{else}\{y := y + 1;\} : \emptyset}\ \text{(T-If)}$$

Passo 9: Verifica del while completo:

$$\frac{\Gamma_2 \vdash x \neq y : \text{bool} \quad \Gamma_2 \vdash \text{if}(x < y)\{...\}\text{else}\{...\} : \emptyset}{\Gamma_2 \vdash \text{while}(x \neq y)\{\text{if}(x < y)\{x := x + 2;\}\text{else}\{y := y + 1;\}\} : \emptyset}\ \text{(T-While)}$$

Passo 10: Verifica della sequenza completa tramite (T-Seq):

$$\frac{\Gamma_0 \vdash \text{int } x = 5; : \{x : \text{int}\} \quad \Gamma_1 \vdash \text{int } y = 6; \text{while...} : \{y : \text{int}\}}{\Gamma_0 \vdash \text{int } x = 5; \text{int } y = 6; \text{while...} : \{x : \text{int}, y : \text{int}\}} \quad \text{(T-Seq)}$$

Il programma e **ben tipato**.

4.2.8 Controllo di tipi e inferenza di tipo

Il sistema di tipi di MAO e un sistema a **controllo di tipi** (type checking): il programmatore dichiara esplicitamente il tipo di ogni variabile e il type checker verifica che l'uso sia coerente con le dichiarazioni. Questa non e l'unica strategia possibile.

Molti linguaggi moderni adottano approcci diversi:

- **Linguaggi senza controllo di tipi** (ad esempio JavaScript): i tipi delle variabili non vengono verificati staticamente; eventuali errori di tipo emergono solo a tempo di esecuzione
- **Linguaggi con inferenza di tipo** (ad esempio Go, Haskell, OCaml): il compilatore *deduce* automaticamente il tipo di una variabile dal contesto in cui viene usata, senza che il programmatore debba dichiararlo esplicitamente

Esempio 4.13 – Inferenza di tipo. In un linguaggio con inferenza di tipo, la dichiarazione

```
x := 5 + 3;
```

permette al compilatore di dedurre che `x` ha tipo `int` dal fatto che `5 + 3` e un'espressione aritmetica il cui risultato e un intero.

4.3 Estensioni del linguaggio

4.3.1 Tipi base aggiuntivi: char e stringhe

4.3.1.1 Caratteri

Il tipo `char` rappresenta singoli simboli, lettere e altri caratteri alfanumerici. In MAO i caratteri possono essere codificati secondo lo standard `ASCII` o `Unicode`. I caratteri *speciali* (come il ritorno a capo o la tabulazione) vengono rappresentati tramite *sequenze di escape*, cioe combinazioni di caratteri che iniziano con il backslash.

Esempio 4.14 – Dichiarazione di caratteri.

```
char lettera = 'R';
char a_capo = '\n';
```

Il valore `'\n'` e la sequenza di escape che rappresenta il carattere di ritorno a capo (*newline*).

4.3.1.2 Stringhe

Le stringhe in MAO sono trattate come array di caratteri, ovvero hanno tipo `char[]`. Questa scelta semplifica la semantica: tutte le operazioni sugli array (accesso per indice, `.length`, assegnamento) si applicano direttamente alle stringhe.

Esempio 4.15 – Stringhe come array di caratteri. La stringa `"Ciao"` e equivalente all'array:

```
char[] saluto = ['C', 'i', 'a', 'o'];
```

Pertanto `saluto.length` restituisce 4 e `saluto[0]` restituisce `'C'`.

4.3.2 Assegnamento multiplo

Molti linguaggi di programmazione permettono di dichiarare o assegnare piu variabili contemporaneamente in un unico comando. Questa funzionalita, detta **assegnamento multiplo**, consente di scrivere codice piu compatto e, in alcuni casi, di realizzare operazioni altrimenti impossibili con assegnamenti singoli.

> **Esempio 4.16 – Dichiarazione multipla.**
>
> `let x, y, z = 6, 7, 42;`
>
> Questa singola istruzione dichiara tre variabili e assegna a ciascuna il valore corrispondente nella lista di destra.

L'assegnamento multiplo e particolarmente utile per lo **scambio di variabili** (swap), che con assegnamenti singoli richiederebbe una variabile temporanea:

Swap tramite assegnamento multiplo

```
x, y := y, x;
```

Tutte le espressioni a destra vengono valutate *prima* che qualsiasi assegnamento abbia luogo. Cio significa che i valori originali di `x` e `y` vengono letti, e solo successivamente scritti nelle posizioni scambiate.

4.3.2.1 Sintassi dell'assegnamento multiplo

Si introducono due nuove categorie sintattiche, LHS (*Left-Hand Side*) e RHS (*Right-Hand Side*):

$$\begin{aligned} \text{LHS} &::= \text{Id} \mid \text{Id}, \text{LHS} \\ \text{RHS} &::= E \mid E, \text{RHS} \end{aligned}$$

I comandi atomici vengono generalizzati per includere la forma multipla:

$$C ::= \ldots \mid T \;\; \text{LHS} = \text{RHS}; \mid \text{LHS} := \text{RHS};$$

4.3.2.2 Type checking per l'assegnamento multiplo

Le variabili libere di LHS e RHS sono definite come:

$$\begin{aligned} \text{fv}(\text{Id}) &= \{\text{Id}\} & \text{fv}(\text{Id}, \text{LHS}) &= \{\text{Id}\} \cup \text{fv}(\text{LHS}) \\ \text{fv}(E) &= \text{fv}(E) & \text{fv}(E, \text{RHS}) &= \text{fv}(E) \cup \text{fv}(\text{RHS}) \end{aligned}$$

La regola di tipo per l'assegnamento multiplo richiede che ogni espressione E_i abbia un tipo coerente con il tipo della variabile corrispondente Id_i. Si noti che ciascuna coppia variabile-espressione viene verificata in modo indipendente: variabili diverse possono avere tipi diversi.

$$\frac{\Gamma(\text{Id}_i) = T_i \quad \Gamma \vdash E_i : T_i \quad \text{per } i = 1..n}{\Gamma \vdash \text{Id}_1, \ldots, \text{Id}_n := E_1, \ldots, E_n; : \emptyset} \quad \text{(T-MultiAssign)}$$

> **Nota.** La regola (T-MultiAssign) si applica all'assegnamento di variabili semplici. Per l'assegnamento a elementi di array all'interno di un assegnamento multiplo (ad esempio `a[0], b[1] := 5, 10;`) si applicano le stesse verifiche di tipo della regola (T-ArrayAssign) per ciascuna posizione del lato sinistro che sia un accesso ad array.

4.3.2.3 Semantica operazionale dell'assegnamento multiplo

La semantica della dichiarazione multipla prevede la valutazione sequenziale di tutte le espressioni, seguita dall'allocazione simultanea delle variabili:

Dichiarazione multipla:

$$\frac{\langle E_i, \rho, \sigma_{i-1}\rangle \Downarrow (v_i, \sigma_i) \text{ per } i = 1..n \quad l_1, ..., l_n \notin \text{dom}(\sigma_n)}{\langle T\ \text{Id}_1, ..., \text{Id}_n = E_1, ..., E_n;, \rho, \sigma\rangle \to \langle \rho[\text{Id}_1 \mapsto l_1]...[\text{Id}_n \mapsto l_n], \sigma_{n[l_1 \mapsto v_1]}...[l_n \mapsto v_n]\rangle}$$

Assegnamento multiplo:

$$\frac{\langle E_i, \rho, \sigma_{i-1}\rangle \Downarrow (v_i, \sigma_i) \text{ per } i = 1..n \quad \rho(\text{Id}_i) = l_i \text{ per } i = 1..n}{\langle \text{Id}_1, ..., \text{Id}_n := E_1, ..., E_n;, \rho, \sigma\rangle \to \langle \rho, \sigma_{n[l_1 \mapsto v_1]}...[l_n \mapsto v_n]\rangle}$$

> **Nota – Semantica dello swap.** Nell'assegnamento multiplo `x, y := y, x`, la semantica garantisce che tutte le espressioni del lato destro vengano valutate *prima* di effettuare gli assegnamenti. Questo significa che `y` e `x` vengono letti con i loro valori originali, e solo dopo i risultati vengono scritti nelle locazioni di `x` e `y` rispettivamente. Lo swap avviene correttamente senza bisogno di variabili temporanee.

4.3.3 Direttiva return

La direttiva `return` permette a un blocco di codice (tipicamente il corpo di una funzione) di restituire un valore al chiamante. Il valore restituito e il risultato della valutazione dell'espressione che segue la parola chiave `return`.

> **Esempio 4.17 – Uso della direttiva return.**
>
> ```
> { count := count + 1; return count; }
> ```
>
> Questo blocco incrementa la variabile `count` e restituisce il suo nuovo valore.

4.3.3.1 Sintassi

La grammatica dei comandi viene estesa con la produzione:

$$C ::= ... \mid \text{return } E;$$

4.3.3.2 Variabili libere

Le variabili libere di un comando `return` sono quelle dell'espressione restituita:

$$\text{fv}(\text{return } E;) = \text{fv}(E)$$

4.3.3.3 Type checking

La regola di tipo verifica che l'espressione restituita sia ben tipata. Il comando `return` non introduce nuove variabili.

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash \text{return } E; : \emptyset} \qquad \text{(T-Return)}$$

> **Nota.** In un sistema di tipi completo, il tipo T dell'espressione restituita dovrebbe essere confrontato con il tipo di ritorno dichiarato nella firma della funzione. In MAO questo controllo viene effettuato dalla regola (T-Fun) o (T-RecFun).

4.3.3.4 Semantica operazionale

L'esecuzione di `return E;` valuta l'espressione E nello stato corrente e produce una terna (v, ρ, σ') che segnala la terminazione con il valore v:

$$\frac{\langle E, \rho, \sigma\rangle \Downarrow (v, \sigma')}{\langle \text{return } E;, \rho, \sigma\rangle \to (v, \rho, \sigma')} \qquad \text{(Return)}$$

4.3.3.5 Propagazione del return nei comandi composti

Quando un comando `return` viene eseguito all'interno di un comando composto (sequenza, condizionale o ciclo), il valore restituito deve **propagarsi verso l'alto** fino al contesto della funzione chiamante. In pratica, l'esecuzione di un `return` interrompe immediatamente l'esecuzione del corpo della funzione e restituisce il valore al chiamante.

Nella semantica big-step di MAO, questa propagazione si realizza nel modo seguente: quando l'esecuzione di un sotto-comando produce una terna (v, ρ, σ') anziche una coppia (ρ', σ'), cio segnala che un `return` e stato eseguito. I comandi composti che lo contengono devono propagare questa terna senza eseguire ulteriori istruzioni.

- **Sequenza**: se nella sequenza C_1C_2 l'esecuzione di C_1 produce (v, ρ', σ'), allora C_2 non viene eseguito e il risultato complessivo e (v, ρ', σ'). Se C_1 termina normalmente e C_2 produce un return, il risultato e quello di C_2.
- **Condizionale**: se il ramo selezionato (then o else) produce (v, ρ', σ'), questo risultato viene propagato.
- **Ciclo while**: se il corpo del while produce (v, ρ', σ'), il ciclo si interrompe e il valore viene propagato.

> **Nota.** La distinzione tra terminazione normale (coppia (ρ', σ')) e terminazione con return (terna (v, ρ', σ')) e il meccanismo chiave che permette al `return` di interrompere il flusso di esecuzione e propagare il valore restituito attraverso qualsiasi livello di annidamento fino alla regola (Call).

4.4 Funzioni

Le funzioni sono un meccanismo di astrazione fondamentale nella programmazione. Permettono di associare un nome a un frammento di codice parametrico, che calcola un valore a partire da uno o piu argomenti. Una volta definita, una funzione puo essere *invocata* (chiamata) piu volte con argomenti diversi, favorendo il riuso del codice e la modularita del programma.

Si distinguono due momenti:

- **Definizione della funzione**: si scrive il codice del corpo della funzione, specificando i *parametri formali* (nomi simbolici che rappresentano gli input)
- **Invocazione (chiamata) della funzione**: si esegue il corpo della funzione con *parametri attuali* (espressioni concrete i cui valori vengono passati ai parametri formali)

> **Esempio 4.18 – Definizione e invocazione di una funzione.**

```
int max(int a, int b){
    int m = a;
    if (b > m) {
        m := b;
    }
    return m;
```

```
}
...
if (max(x, y) < 10) {
    z := max(x + 2, y * 3);
} else {
    z := max(x / 10, y - 10);
}
```

La funzione `max` e definita con due parametri formali `a` e `b`. Viene poi invocata tre volte con parametri attuali diversi.

4.4.1 Corrispondenza tra parametri formali e attuali

Quando si invoca una funzione, i parametri attuali devono corrispondere ai parametri formali **in numero e in tipo**. La corrispondenza e *posizionale*: il primo parametro attuale viene associato al primo parametro formale, il secondo al secondo, e cosi via.

4.4.2 Passaggio di parametri per valore

Il **passaggio per valore** e la modalita di default in MAO per i tipi scalari (`int`, `bool`, `char`). Ogni parametro formale viene inizializzato con una *copia* del valore del corrispondente parametro attuale. Di conseguenza, eventuali modifiche ai parametri formali all'interno del corpo della funzione *non influenzano* i parametri attuali nel contesto del chiamante.

4.4.3 Passaggio di parametri per riferimento (array)

Quando un array viene passato come parametro attuale, il valore copiato nel parametro formale e l'*indirizzo base* l_b dell'array. Si tratta tecnicamente ancora di un passaggio per valore (si copia il riferimento), ma il risultato pratico e un **passaggio per riferimento implicito**: la funzione e il chiamante condividono lo stesso array in memoria, quindi le modifiche agli elementi dell'array effettuate nel corpo della funzione sono visibili anche nel contesto del chiamante.

> **Nota.** Questa e una conseguenza diretta del meccanismo di aliasing descritto in precedenza. Poiche la variabile di tipo array contiene un riferimento e non i dati stessi, copiare la variabile equivale a creare un alias.

4.4.4 Sintassi delle funzioni

4.4.4.1 Dichiarazione

La sintassi della dichiarazione di funzione e la seguente:

$$C ::= \ldots \mid T_R \ \ \mathrm{Id}(T_1 \ \mathrm{Id}_1, \ldots, T_n \ \mathrm{Id}_n)\{C\}$$

dove T_R e il **tipo di ritorno** (puo essere `void` per funzioni che non restituiscono un valore), Id e il nome della funzione, $T_i \ \mathrm{Id}_i$ sono le coppie tipo-nome dei parametri formali, e C e il corpo della funzione.

4.4.4.2 Invocazione

La sintassi della chiamata di funzione e:

$$E ::= \ldots \mid \mathrm{Id}(E_1, \ldots, E_n)$$

dove le espressioni $E_1, \ldots, E_n$ sono i parametri attuali.

4.4.4.3 Tipo di una funzione

Il tipo di una funzione viene rappresentato come un tipo freccia:

$$(T_1, ..., T_n) \to T_R$$

dove $(T_1, ..., T_n)$ sono i tipi dei parametri formali e T_R e il tipo di ritorno.

4.4.4.4 Variabili libere di una funzione

Le variabili libere nel corpo di una funzione escludono i parametri formali, che sono dichiarati nell'intestazione:

$$\text{fv}(T_R \ \ \text{Id}(T_1 \ \text{Id}_1, ..., T_n \ \text{Id}_n)\{C\}) = \text{fv}(C) \setminus \{\text{Id}_1, ..., \text{Id}_n\}$$

4.4.5 Type checking delle funzioni

4.4.5.1 Dichiarazione di funzione

La regola di tipo per la dichiarazione di una funzione verifica che il corpo C sia ben tipato in un ambiente esteso con i parametri formali. La dichiarazione introduce nell'ambiente l'associazione tra il nome della funzione e il suo tipo freccia.

$$\frac{\Gamma \cup \{\text{Id}_1 : T_1, ..., \text{Id}_n : T_n\} \vdash C : \Gamma'}{\Gamma \vdash T_R \ \ \text{Id}(T_1 \ \text{Id}_1, ..., T_n \ \text{Id}_n)\{C\} : \{\text{Id} : (T_1, ..., T_n) \to T_R\}} \quad \text{(T-Fun)}$$

> **Nota.** Si osservi che il corpo della funzione viene verificato nell'ambiente $\Gamma \cup \{\text{Id}_1 : T_1, ..., \text{Id}_n : T_n\}$, in cui i parametri formali sono disponibili con i loro tipi dichiarati. L'ambiente Γ del chiamante resta invariato tranne per l'aggiunta del tipo della funzione.

4.4.5.2 Invocazione di funzione

La regola per l'invocazione verifica che la funzione sia presente nell'ambiente con un tipo freccia compatibile e che i tipi dei parametri attuali corrispondano (posizionalmente) ai tipi dei parametri formali.

$$\frac{\Gamma(\text{Id}) = (T_1, ..., T_n) \to T_R \quad \Gamma \vdash E_1 : T_1 \quad ... \quad \Gamma \vdash E_n : T_n}{\Gamma \vdash \text{Id}(E_1, ..., E_n) : T_R} \quad \text{(T-Call)}$$

4.4.6 Semantica operazionale delle funzioni

4.4.6.1 Dichiarazione di funzione

La dichiarazione di una funzione non esegue il corpo, ma memorizza nell'ambiente una **chiusura** (*closure*), ovvero una terna composta dai nomi dei parametri formali, dal corpo della funzione e dall'ambiente al momento della dichiarazione:

$$\langle T_R \ \ \text{Id}(T_1 \ \text{Id}_1, ..., T_n \ \text{Id}_n)\{C\}, \rho, \sigma \rangle \to \langle \rho[\text{Id} \mapsto (\text{Id}_1, ..., \text{Id}_n, C, \rho)], \sigma \rangle$$

> **Nota – Chiusura (closure).** La chiusura cattura l'ambiente ρ al momento della dichiarazione. Questo e essenziale per lo **scoping statico** (o *lessicale*): quando la funzione verra invocata, le variabili libere nel corpo saranno risolte nell'ambiente della dichiarazione, non in quello della chiamata. Senza la chiusura, una funzione definita in un certo contesto e chiamata in un altro potrebbe accedere a variabili diverse da quelle previste dal programmatore.

4.4.6.2 Chiamata di funzione

La chiamata di funzione si articola nei seguenti passi:

1. Si recupera la chiusura della funzione dall'ambiente del chiamante

2. Si valutano i parametri attuali $E_1, ..., E_n$ da sinistra a destra
3. Si allocano nuove locazioni $l_1, ..., l_n$ per i parametri formali
4. Si costruisce un nuovo ambiente ρ' estendendo l'ambiente della chiusura ρ_f con le associazioni tra parametri formali e locazioni
5. Si esegue il corpo C nel nuovo ambiente ρ'
6. Il valore restituito dal `return` diventa il risultato della chiamata

$$\frac{\begin{array}{c}\rho(\mathrm{Id}) = (\mathrm{Id}_1, ..., \mathrm{Id}_n, C, \rho_f) \\ \langle E_i, \rho, \sigma_{i-1}\rangle \Downarrow (v_i, \sigma_i) \text{ per } i = 1..n \\ l_1, ..., l_n \notin \mathrm{dom}(\sigma_n) \\ \rho' = \rho_{f[\mathrm{Id}_1 \mapsto l_1]}...[\mathrm{Id}_n \mapsto l_n] \\ \sigma' = \sigma_{n[l_1 \mapsto v_1]}...[l_n \mapsto v_n] \\ \langle C, \rho', \sigma'\rangle \to (v_r, \rho'', \sigma'')\end{array}}{\langle \mathrm{Id}(E_1, ..., E_n), \rho, \sigma\rangle \Downarrow (v_r, \sigma'')} \quad \text{(Call)}$$

> **Nota – Scoping statico.** Si osservi che il nuovo ambiente ρ' e costruito a partire da ρ_f (l'ambiente catturato nella chiusura), **non** da ρ (l'ambiente del chiamante). Questo implementa lo scoping statico: le variabili libere nel corpo della funzione vengono risolte nel contesto in cui la funzione e stata *definita*, non in quello in cui viene *chiamata*.

4.5 Ricorsione

In molti linguaggi di programmazione e ammessa la possibilita di definire **funzioni ricorsive**, cioe funzioni che invocano se stesse (direttamente o indirettamente) nel proprio corpo. La ricorsione e un meccanismo potente che permette di risolvere problemi definiti in modo induttivo, decomponendoli in sottoproblemi della stessa natura ma di dimensione ridotta.

La chiamata ricorsiva puo essere:
- **Diretta**: la funzione f contiene nel proprio corpo una chiamata a f stessa
- **Indiretta**: la funzione f chiama una funzione g, che a sua volta chiama f (oppure tramite una catena piu lunga di chiamate intermedie)

4.5.1 Variabili libere nelle funzioni ricorsive

Per supportare la ricorsione, la definizione di variabili libere di una funzione viene modificata includendo anche il nome della funzione stessa tra le variabili escluse:

$$\mathrm{fv}(T_R\ \ \mathrm{Id}(T_1\ \mathrm{Id}_1, ..., T_n\ \mathrm{Id}_n)\{C\}) = \mathrm{fv}(C) \setminus \{\mathrm{Id}, \mathrm{Id}_1, ..., \mathrm{Id}_n\}$$

In questo modo il nome della funzione non e considerato una variabile libera nel corpo, anche se vi compare come chiamata ricorsiva. Senza questa modifica, il type checker richiederebbe che `Id` fosse gia presente nell'ambiente Γ prima della dichiarazione, rendendo impossibile la ricorsione.

4.5.2 Type checking delle funzioni ricorsive

La regola di tipo per le funzioni ricorsive differisce da (T-Fun) per il fatto che l'ambiente in cui viene verificato il corpo include anche l'associazione tra il nome della funzione e il suo tipo freccia. Questo permette al type checker di verificare le chiamate ricorsive:

$$\frac{\Gamma \cup \{\mathrm{Id} : (T_1, ..., T_n) \to T_R\} \cup \{\mathrm{Id}_1 : T_1, ..., \mathrm{Id}_n : T_n\} \vdash C : \Gamma'}{\Gamma \vdash T_R\ \ \mathrm{Id}(T_1\ \mathrm{Id}_1, ..., T_n\ \mathrm{Id}_n)\{C\} : \{\mathrm{Id} : (T_1, ..., T_n) \to T_R\}} \quad \text{(T-RecFun)}$$

Nota – Differenza tra T-Fun e T-RecFun. La differenza chiave e nell'ambiente usato per verificare il corpo C. In (T-Fun), l'ambiente e $\Gamma \cup \{\mathrm{Id}_1 : T_1, ..., \mathrm{Id}_n : T_n\}$; in (T-RecFun), e $\Gamma \cup \{\mathrm{Id} : (T_1, ..., T_n) \to T_R\} \cup \{\mathrm{Id}_1 : T_1, ..., \mathrm{Id}_n : T_n\}$. La presenza del binding $\mathrm{Id} : (T_1, ..., T_n) \to T_R$ permette di verificare le chiamate ricorsive nel corpo della funzione come normali invocazioni tramite la regola (T-Call).

4.5.3 Semantica operazionale delle funzioni ricorsive

Nella semantica operazionale, la ricorsione funziona perche la chiusura di una funzione ricorsiva include il nome della funzione stessa nell'ambiente catturato. Al momento della dichiarazione, l'ambiente ρ viene esteso con il binding della funzione *prima* di costruire la chiusura:

$$\langle T_R \;\; \mathrm{Id}(T_1 \; \mathrm{Id}_1, ..., T_n \; \overline{\mathrm{Id}}_n)\{C\}, \rho, \sigma\rangle \to \langle \rho', \sigma\rangle$$

dove $\rho' = \rho[\mathrm{Id} \mapsto (\mathrm{Id}_1, ..., \mathrm{Id}_n, C, \rho')]$

Nota – Autoreferenza nella chiusura. Si osservi la natura circolare della definizione: l'ambiente ρ' contenuto nella chiusura include il binding di Id alla chiusura stessa. Questa autoreferenza e cio che rende possibile la ricorsione a livello semantico. Quando il corpo C viene eseguito durante una chiamata, il nome della funzione e presente nell'ambiente ρ' e punta alla stessa chiusura, permettendo di effettuare chiamate ricorsive. Senza questo meccanismo, il corpo della funzione non troverebbe il binding per Id nell'ambiente e la chiamata ricorsiva fallirebbe. La regola (Call) non necessita di modifiche: funziona identicamente per funzioni ricorsive e non ricorsive, perche la ricorsione e interamente gestita dalla struttura della chiusura.

4.5.4 Esempi completi

Esempio 4.19 – Type checking della funzione abs. Verifichiamo che la funzione `abs` (valore assoluto) sia ben tipata:

```
int abs(int n) {
  int m = n;
  if (m < 0) { m := -m; }
  return m;
}
```

Vogliamo dimostrare: $\Gamma \vdash \mathrm{abs} : (\mathrm{int}) \to \mathrm{int}$

Passo 1: Costruzione dell'ambiente per il corpo.

Secondo la regola (T-Fun), il corpo deve essere verificato nell'ambiente:

$$\Gamma' = \Gamma \cup \{n : \mathrm{int}\}$$

Passo 2: Type checking di `int m = n;` in Γ'

$$\dfrac{\dfrac{\Gamma'(n) = \mathrm{int}}{\Gamma' \vdash n : \mathrm{int}}}{\Gamma' \vdash \mathrm{int} \;\; m = n; : \{m : \mathrm{int}\}} \quad \text{(T-Decl)}$$

Ambiente aggiornato: $\Gamma'' = \Gamma' \cup \{m : \mathrm{int}\} = \Gamma \cup \{n : \mathrm{int}, m : \mathrm{int}\}$

Passo 3: Type checking della guardia `m < 0` in Γ''

$$\frac{\Gamma''(m) = \text{int} \qquad 0 \in \mathbb{Z}}{\dfrac{\Gamma'' \vdash m : \text{int} \qquad \Gamma'' \vdash 0 : \text{int}}{\Gamma'' \vdash m < 0 : \text{bool}}\ \text{(T-Cop)}}$$

Passo 4: Type checking della negazione `-m` in Γ''

L'operatore unario meno richiede un operando intero e produce un intero:

$$\frac{\Gamma'' \vdash m : \text{int}}{\Gamma'' \vdash -m : \text{int}}\ \text{(T-Neg)}$$

Passo 5: Type checking dell'assegnamento `m := -m;` in Γ''

$$\frac{\Gamma''(m) = \text{int} \quad \Gamma'' \vdash -m : \text{int}}{\Gamma'' \vdash m := -m; : \emptyset}\ \text{(T-Assign)}$$

Passo 6: Type checking del blocco `{ m := -m; }` in Γ''

$$\frac{\Gamma'' \vdash m := -m; : \emptyset}{\Gamma'' \vdash \{m := -m;\} : \emptyset}\ \text{(T-Block)}$$

Passo 7: Type checking del condizionale (con else implicito uguale a skip)

$$\frac{\Gamma'' \vdash m < 0 : \text{bool} \quad \Gamma'' \vdash \{m := -m;\} : \emptyset \quad \Gamma'' \vdash \text{skip}; : \emptyset}{\Gamma'' \vdash \text{if}(m < 0)\{m := -m;\} : \emptyset}\ \text{(T-If)}$$

Passo 8: Type checking di `return m;` in Γ''

$$\frac{\Gamma''(m) = \text{int}}{\dfrac{\Gamma'' \vdash m : \text{int}}{\Gamma'' \vdash \text{return } m; : \emptyset}\ \text{(T-Return)}}$$

Il tipo restituito (`int`) corrisponde al tipo di ritorno dichiarato nella firma.

Passo 9: Composizione sequenziale del corpo della funzione

$$\frac{\Gamma' \vdash \text{int } m = n; : \{m : \text{int}\} \quad \Gamma'' \vdash \text{if}(...)\{...\}\ \text{return } m; : \emptyset}{\Gamma' \vdash \text{int } m = n; \text{if}(m < 0)\{m := -m;\}\ \text{return } m; : \{m : \text{int}\}}\ \text{(T-Seq)}$$

Passo 10: Applicazione della regola (T-Fun)

$$\frac{\Gamma \cup \{n : \text{int}\} \vdash \text{int } m = n; \text{if}(m < 0)\{m := -m;\}\ \text{return } m; : \{m : \text{int}\}}{\Gamma \vdash \text{int} \quad \text{abs}(\text{int } n)\{...\} : \{\text{abs} : (\text{int}) \to \text{int}\}}\ \text{(T-Fun)}$$

Quindi $\Gamma \vdash \text{abs} : (\text{int}) \to \text{int}$ come richiesto.

Esempio 4.20 – Type checking di una funzione ricorsiva su array. Verifichiamo il type checking della funzione ricorsiva `azzera`, che cerca l'elemento `k` nell'array `a` a partire dalla posizione `p` e, se lo trova, lo sostituisce con 0:

```
bool azzera(int[] a, int k, int p) {
  bool res = false;
  if ((p >= 0) && (p < a.length)) {
    if (a[p] == k) { a[p] := 0; res := true; }
    res := azzera(a, k, p+1);
  }
  return res;
}
```

Passo 1: Poiche la funzione e ricorsiva, usiamo la regola (T-RecFun).

L'ambiente per il corpo deve includere il tipo della funzione stessa (per permettere la chiamata ricorsiva) e i parametri formali:

$$\Gamma' = \Gamma \cup \{\text{azzera} : (\text{int}[], \text{int}, \text{int}) \to \text{bool}\} \cup \{a : \text{int}[], k : \text{int}, p : \text{int}\}$$

Passo 2: Type checking di `bool res = false;` in Γ'

$$\frac{\dfrac{}{\Gamma' \vdash \text{false} : \text{bool}}}{\Gamma' \vdash \text{bool} \quad \text{res} = \text{false}; : \{\text{res} : \text{bool}\}} \quad \text{(T-Decl)}$$

Ambiente aggiornato: $\Gamma'' = \Gamma' \cup \{\text{res} : \text{bool}\}$

Passo 3: Type checking della guardia `(p >= 0) && (p < a.length)` in Γ''

Verifica di `p >= 0`:

$$\frac{\dfrac{\Gamma''(p) = \text{int} \quad 0 \in \mathbb{Z}}{\Gamma'' \vdash p : \text{int}} \quad \Gamma'' \vdash 0 : \text{int}}{\Gamma'' \vdash p \geq 0 : \text{bool}} \quad \text{(T-Cop)}$$

Verifica di `a.length`:

$$\frac{\dfrac{\Gamma''(a) = \text{int}[]}{\Gamma'' \vdash a : \text{int}[]}}{\Gamma'' \vdash a.\text{length} : \text{int}} \quad \text{(T-Length)}$$

Verifica di `p < a.length`:

$$\frac{\Gamma'' \vdash p : \text{int} \quad \Gamma'' \vdash a.\text{length} : \text{int}}{\Gamma'' \vdash p < a.\text{length} : \text{bool}} \quad \text{(T-Cop)}$$

Congiunzione logica:

$$\frac{\Gamma'' \vdash p \geq 0 : \text{bool} \quad \Gamma'' \vdash p < a.\text{length} : \text{bool}}{\Gamma'' \vdash (p \geq 0) \wedge (p < a.\text{length}) : \text{bool}} \quad \text{(T-Lop)}$$

Passo 4: Type checking dell'accesso `a[p]` in Γ''

$$\frac{\Gamma'' \vdash a : \text{int}[] \quad \Gamma'' \vdash p : \text{int}}{\Gamma'' \vdash a[p] : \text{int}} \quad \text{(T-ArrayAccess)}$$

Passo 5: Type checking della condizione `a[p] == k`

$$\frac{\Gamma'' \vdash a[p] : \text{int} \quad \Gamma'' \vdash k : \text{int}}{\Gamma'' \vdash a[p] == k : \text{bool}} \quad \text{(T-Cop)}$$

Passo 6: Type checking dell'assegnamento `a[p] := 0;`

$$\frac{\Gamma'' \vdash a : \text{int}[] \quad \Gamma'' \vdash p : \text{int} \quad \Gamma'' \vdash 0 : \text{int}}{\Gamma'' \vdash a[p] := 0; : \emptyset} \quad \text{(T-ArrayAssign)}$$

Passo 7: Type checking dell'assegnamento `res := true;`

$$\frac{\Gamma''(\text{res}) = \text{bool} \quad \Gamma'' \vdash \text{true} : \text{bool}}{\Gamma'' \vdash \text{res} := \text{true}; : \emptyset} \quad \text{(T-Assign)}$$

Passo 8: Type checking della chiamata ricorsiva `azzera(a, k, p+1)`

Prima verifichiamo `p+1`:

$$\frac{\Gamma'' \vdash p : \text{int} \quad \Gamma'' \vdash 1 : \text{int}}{\Gamma'' \vdash p + 1 : \text{int}} \quad \text{(T-Aop)}$$

Poi la chiamata, usando il tipo di `azzera` presente in Γ'' grazie alla regola (T-RecFun):

$$\frac{\Gamma''(\text{azzera}) = (\text{int}[], \text{int}, \text{int}) \to \text{bool} \quad \Gamma'' \vdash a : \text{int}[] \quad \Gamma'' \vdash k : \text{int} \quad \Gamma'' \vdash p + 1 : \text{int}}{\Gamma'' \vdash \text{azzera}(a, k, p + 1) : \text{bool}} \quad \text{(T-Call)}$$

Passo 9: Type checking dell'assegnamento `res := azzera(a, k, p+1);`

$$\frac{\Gamma''(\text{res}) = \text{bool} \quad \Gamma'' \vdash \text{azzera}(a, k, p + 1) : \text{bool}}{\Gamma'' \vdash \text{res} := \text{azzera}(a, k, p + 1); : \emptyset} \quad \text{(T-Assign)}$$

Passo 10: Type checking di `return res;`

$$\frac{\Gamma'' \vdash \text{res} : \text{bool}}{\Gamma'' \vdash \text{return res}; : \emptyset} \quad \text{(T-Return)}$$

Il tipo restituito (`bool`) corrisponde al tipo di ritorno dichiarato nella firma.

Passo 11: Applicazione della regola (T-RecFun)

$$\frac{\Gamma \cup \{\text{azzera} : (\text{int}[], \text{int}, \text{int}) \to \text{bool}\} \cup \{a : \text{int}[], k : \text{int}, p : \text{int}\} \vdash C : \Gamma'''}{\Gamma \vdash \text{bool} \quad \text{azzera}(\text{int}[]\ a, \text{int}\ \ k, \text{int}\ \ p)\{C\} : \{\text{azzera} : (\text{int}[], \text{int}, \text{int}) \to \text{bool}\}}$$

dove C rappresenta il corpo completo della funzione.

La funzione e **ben tipata** e ha tipo $(\text{int}[], \text{int}, \text{int}) \to \text{bool}$.

5 Complessità Computazionale

5.1 Modello di calcolo e costo computazionale

In questo capitolo introduciamo gli strumenti fondamentali per analizzare l'efficienza degli algoritmi. Dato un problema computazionale, esistono in generale molteplici algoritmi che lo risolvono: il nostro obiettivo è confrontarli in modo rigoroso, determinando quale sia il più efficiente in termini di risorse utilizzate. Per fare ciò, abbiamo bisogno di un modello di calcolo di riferimento e di un linguaggio matematico per esprimere il costo degli algoritmi.

5.1.1 Modello di calcolo RAM

Per poter contare le operazioni eseguite da un algoritmo, occorre fissare un modello di calcolo. Il modello adottato nel corso è il **modello RAM** (Random Access Machine), che formalizza un calcolatore idealizzato.

> **Definizione 5.1 – Modello RAM a costo unitario.** Nel modello RAM a costo unitario, le seguenti operazioni hanno ciascuna costo costante (pari a 1):
> - **Operazioni aritmetiche**: $+, -, \times, \div, \%$
> - **Operazioni di confronto**: $<, >, \leq, \geq, ==, \neq$
> - **Operazioni logiche**: $\wedge, \vee, \neg$
> - **Operazioni di trasferimento**: lettura e scrittura in memoria, assegnamento
> - **Operazioni di controllo**: `return`, chiamata a funzione, salto condizionale
>
> La memoria è ad accesso casuale: l'accesso a qualsiasi cella ha costo costante, indipendentemente dal suo indirizzo.

> **Nota.** Il modello RAM a costo unitario è una semplificazione: in un calcolatore reale, il costo di un'operazione può dipendere dalla dimensione degli operandi (ad esempio, moltiplicare due numeri di 1000 cifre è più costoso che moltiplicare due numeri di una cifra). Tuttavia, per gli scopi del corso, questa approssimazione è adeguata e consente di concentrarsi sugli aspetti strutturali degli algoritmi.

5.1.2 Costo computazionale di un algoritmo

Dato un algoritmo A e un input di dimensione n, definiamo il **costo computazionale** come la quantità di risorse richieste dalla sua esecuzione.

> **Definizione 5.2 – Costo in tempo.** Il **costo in tempo** $T_A(n)$ di un algoritmo A è il numero di operazioni elementari (passi) che A esegue su un input di dimensione n nel modello RAM.

> **Definizione 5.3 – Costo in spazio.** Il **costo in spazio** $S_A(n)$ di un algoritmo A è il numero di celle di memoria utilizzate durante l'esecuzione di A su un input di dimensione n, incluse quelle occupate dall'input stesso.

In generale, il costo in tempo di un algoritmo non dipende solo dalla dimensione dell'input, ma anche dalla specifica istanza. Per questo si distinguono tre casi.

> **Definizione 5.4 – Caso ottimo, pessimo e medio.** Sia I_n l'insieme di tutte le istanze di dimensione n per un dato problema. Il costo in tempo di un algoritmo A si descrive come:
> - **Caso ottimo**: $T_A^{\text{best}}(n) = \min_{I \in I_n} T_A(I)$ – il minimo numero di operazioni su tutte le istanze di dimensione n.
> - **Caso pessimo**: $T_A^{\text{worst}}(n) = \max_{I \in I_n} T_A(I)$ – il massimo numero di operazioni su tutte le istanze di dimensione n.
> - **Caso medio**: $T_A^{\text{avg}}(n) = \sum_{I \in I_n} P(I) \cdot T_A(I)$ – il costo atteso, dove $P(I)$ è la probabilità dell'istanza I.

Nell'analisi degli algoritmi ci concentreremo principalmente sul **caso pessimo**, poiché fornisce una garanzia sul costo massimo. Il caso medio è spesso più informativo nella pratica, ma richiede ipotesi sulla distribuzione degli input.

A parità di complessità in tempo, si cerca di minimizzare anche la complessità in spazio.

5.1.3 Esempio: Minimo in un vettore

Consideriamo il problema di trovare il valore minimo in un array.

Input: array $A[1..n]$ di interi.
Output: il valore minimo m tale che $m = A[i]$ per qualche $i \in \{1, ..., n\}$ e $m \leq A[j]$ per ogni $j \in \{1, ..., n\}$.

Minimo di un array

```
int min(int[] A, int n){
    int min = A[1];
    int i = 2;
    while(i <= n){
        if(A[i] < min){
            min := A[i];
        }
        i := i + 1;
    }
    return min;
}
```

Analisi del costo. Tutte le operazioni nel corpo del ciclo (`if`, confronto, eventuale assegnamento, incremento) hanno costo costante nel modello RAM. Il ciclo `while` viene eseguito esattamente $n - 1$ volte, indipendentemente dai valori contenuti nell'array. Il costo totale è quindi:

$$T(n) = c_1 + (n-1) \cdot c_2 + c_3$$

dove c_1, c_3 sono costanti per le operazioni fuori dal ciclo e c_2 è il costo costante di ciascuna iterazione. La complessità in tempo è **lineare** nella dimensione n dell'input. Si noti che in questo caso il costo non dipende dalla specifica istanza: caso ottimo, pessimo e medio coincidono.

5.1.4 Esempio: Ricerca di un elemento

Consideriamo il problema della ricerca lineare (o sequenziale).

Input: array $A[1..n]$ di interi, intero k.
Output: il minimo indice i tale che $A[i] = k$, oppure -1 se $k \notin A$.

Ricerca lineare (CercaK)

```
int cercaK(int[] A, int n, int k){
    int i = 1;
    bool trovato = false;
    while((!trovato) && (i <= n)){
        if(A[i] == k){
            trovato := true;
        } else {
            i := i + 1;
        }
    }
    if(trovato){
        return i;
    } else {
        return -1;
    }
}
```

Analisi del costo. A differenza dell'esempio precedente, il numero di iterazioni dipende dalla posizione di k nell'array:

- **Caso ottimo**: $A[1] = k$, il ciclo esegue una sola iterazione. Il costo è $T^{\text{best}(n)} = \Theta(1)$ (costante).
- **Caso pessimo**: $k \notin A$, il ciclo scorre l'intero array senza trovare k. Il costo è $T^{\text{worst}(n)} = \Theta(n)$ (lineare).
- **Caso medio**: assumendo che k sia presente e che ciascuna posizione sia equiprobabile, il numero medio di confronti è $\frac{n+1}{2}$, dunque $T^{\text{avg}(n)} = \Theta(n)$.

Questo esempio mostra come lo stesso algoritmo possa avere costi significativamente diversi a seconda dell'istanza, motivando la distinzione tra caso ottimo e pessimo.

5.1.5 Complessità asintotica

L'obiettivo dell'analisi di complessità non è calcolare il numero esatto di operazioni, ma determinare **l'ordine di grandezza** della funzione di costo $T(n)$ al crescere di n. Si trascurano:

- le **costanti moltiplicative**, che dipendono dal modello di calcolo e dall'implementazione;
- i **termini di ordine inferiore**, che diventano trascurabili per n grande.

> **Esempio 5.1 – Complessità lineare.**
>
> $$T(n) = 3n + 2$$
>
> Il termine dominante è $3n$. Trascurando la costante moltiplicativa 3 e il termine di ordine inferiore 2, si conclude che $T(n)$ ha ordine di grandezza **lineare**.

> **Esempio 5.2 – Complessità quadratica.**
>
> $$T(n) = 8n^2 + \log n + 4$$
>
> Il termine dominante è $8n^2$. I termini $\log n$ e 4 sono di ordine inferiore e vengono trascurati. La complessità è **quadratica**.

Per formalizzare queste nozioni, introduciamo la **notazione asintotica**.

5.1.6 Notazione Θ – Limite asintotico stretto

Definizione 5.5 – Θ (Theta). Sia $g(n)$ una funzione. L'insieme $\Theta(g(n))$ è definito come:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, \quad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Se $f(n) \in \Theta(g(n))$, si dice che $g(n)$ è un **limite asintotico stretto** per $f(n)$.

Si scrive $f(n) = \Theta(g(n))$ (con abuso di notazione, poiché $\Theta(g(n))$ è un insieme).

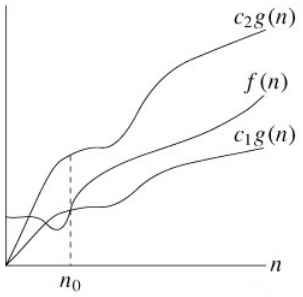


Figura 1: Rappresentazione grafica di $\Theta(g(n))$: per $n \geq n_0$, la funzione $f(n)$ è compresa tra $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$.

Intuitivamente, $f(n) = \Theta(g(n))$ significa che, per n sufficientemente grande, $f(n)$ cresce **allo stesso ritmo** di $g(n)$, a meno di costanti moltiplicative.

5.1.7 Notazione O – Limite asintotico superiore

Definizione 5.6 – O (O grande). Sia $g(n)$ una funzione. L'insieme $O(g(n))$ è definito come:

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0 : \forall n \geq n_0, \quad 0 \leq f(n) \leq c \cdot g(n)\}$$

Se $f(n) \in O(g(n))$, si dice che $g(n)$ è un **limite asintotico superiore** per $f(n)$.

Si scrive $f(n) = O(g(n))$.

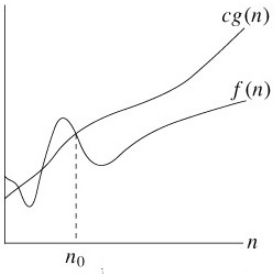


Figura 2: Rappresentazione grafica di $O(g(n))$: per $n \geq n_0$, la funzione $f(n)$ non supera $c \cdot g(n)$.

La notazione O fornisce un **maggiorante** alla crescita di $f(n)$. Si utilizza tipicamente per esprimere il costo nel caso pessimo.

5.1.8 Notazione Ω – Limite asintotico inferiore

Definizione 5.7 – Ω (Omega grande). Sia $g(n)$ una funzione. L'insieme $\Omega(g(n))$ è definito come:

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 : \forall n \geq n_0, \quad 0 \leq c \cdot g(n) \leq f(n)\}$$

Se $f(n) \in \Omega(g(n))$, si dice che $g(n)$ è un **limite asintotico inferiore** per $f(n)$.

Si scrive $f(n) = \Omega(g(n))$.

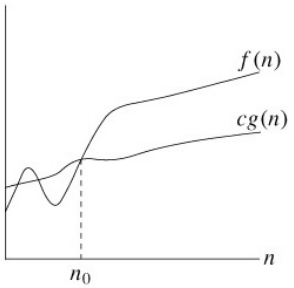


Figura 3: Rappresentazione grafica di $\Omega(g(n))$: per $n \geq n_0$, la funzione $f(n)$ è sempre almeno $c \cdot g(n)$.

La notazione Ω fornisce un **minorante** alla crescita di $f(n)$. Si utilizza tipicamente per esprimere il costo nel caso ottimo o per dimostrare limiti inferiori.

5.1.9 Relazione tra le notazioni

Teorema 5.1 – Relazione tra Θ, O e Ω. Per ogni coppia di funzioni $f(n)$ e $g(n)$:

$$f(n) = \Theta(g(n)) \quad \Leftrightarrow \quad f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

Dimostrazione. ($\Rightarrow$) Se $f(n) = \Theta(g(n))$, allora esistono $c_1, c_2, n_0 > 0$ tali che per ogni $n \geq n_0$:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

La disuguaglianza $f(n) \leq c_2 \cdot g(n)$ implica $f(n) = O(g(n))$ con $c = c_2$. La disuguaglianza $c_1 \cdot g(n) \leq f(n)$ implica $f(n) = \Omega(g(n))$ con $c = c_1$.

($\Leftarrow$) Se $f(n) = O(g(n))$, esiste $c_2, n_1 > 0$ con $f(n) \leq c_2 \cdot g(n)$ per $n \geq n_1$. Se $f(n) = \Omega(g(n))$, esiste $c_1, n_2 > 0$ con $c_1 \cdot g(n) \leq f(n)$ per $n \geq n_2$. Prendendo $n_0 = \max(n_1, n_2)$, entrambe le disuguaglianze valgono simultaneamente per $n \geq n_0$, ovvero $f(n) = \Theta(g(n))$. ■

5.1.10 Proprietà della notazione asintotica

Le notazioni Θ, O e Ω godono di proprietà algebriche che ne facilitano l'uso. Per ciascuna proprietà riportiamo la formulazione e, ove opportuno, una giustificazione intuitiva.

Teorema 5.2 – Transitività. Se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, allora $f(n) = \Theta(h(n))$.

Analogamente per O e Ω:
- $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

Dimostrazione. Dimostriamo il caso O. Per ipotesi, esistono c_1, n_1 con $f(n) \leq c_1 \cdot g(n)$ per $n \geq n_1$, e c_2, n_2 con $g(n) \leq c_2 \cdot h(n)$ per $n \geq n_2$. Per $n \geq \max(n_1, n_2)$:

$$f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

Ponendo $c = c_1 \cdot c_2$ e $n_0 = \max(n_1, n_2)$, si ha $f(n) = O(h(n))$. Le dimostrazioni per Ω e Θ sono analoghe. ■

Teorema 5.3 – Riflessività. Per ogni funzione $f(n)$:

$$f(n) = \Theta(f(n)), \quad f(n) = O(f(n)), \quad f(n) = \Omega(f(n))$$

Basta prendere $c_1 = c_2 = c = 1$ e $n_0 = 1$ nelle rispettive definizioni.

Teorema 5.4 – Simmetria.

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

La simmetria vale **solo** per Θ. Non vale per O e Ω.

Teorema 5.5 – Simmetria trasposta.

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

Intuitivamente: dire che f cresce al più come g equivale a dire che g cresce almeno come f.

Nota – Analogia con le relazioni d'ordine. Le notazioni asintotiche si possono interpretare come relazioni d'ordine tra funzioni:

- $f(n) = O(g(n))$ corrisponde a $f \leq g$ (informalmente)
- $f(n) = \Omega(g(n))$ corrisponde a $f \geq g$
- $f(n) = \Theta(g(n))$ corrisponde a $f \approx g$

Le proprietà di transitività, riflessività e simmetria ricalcano quelle delle relazioni $\leq$, $\geq$ e $=$.

5.1.11 Esercizi sulla notazione asintotica

Esempio 5.3 – Dimostrare che $3n^2 - 2n - 1 \in \Theta(n^2)$. Dobbiamo trovare costanti $c_1, c_2 > 0$ e $n_0 \geq 1$ tali che:

$$\forall n \geq n_0 : \quad c_1 \cdot n^2 \leq 3n^2 - 2n - 1 \leq c_2 \cdot n^2$$

Limite superiore (O): Dimostriamo che $3n^2 - 2n - 1 \leq c_2 \cdot n^2$.

Per $n \geq 1$: $\quad 3n^2 - 2n - 1 \leq 3n^2$

Quindi con $c_2 = 3$ il limite superiore è verificato per ogni $n \geq 1$.

Limite inferiore (Ω): Dimostriamo che $c_1 \cdot n^2 \leq 3n^2 - 2n - 1$.

Riscriviamo: $\quad 3n^2 - 2n - 1 \geq c_1 \cdot n^2$

Dividendo per n^2 (per $n \geq 1$): $\quad 3 - \frac{2}{n} - \frac{1}{n^2} \geq c_1$

Per $n \geq 2$:

- $\frac{2}{n} \leq 1$
- $\frac{1}{n^2} \leq \frac{1}{4}$

Quindi: $\quad 3 - 1 - \frac{1}{4} = \frac{7}{4} \geq c_1$

Scegliendo $c_1 = 1$, verifichiamo per $n = 2$:

$$3(4) - 2(2) - 1 = 12 - 4 - 1 = 7 \geq 1 \cdot 4 = 4 \quad \checkmark$$

Conclusione: Con $c_1 = 1$, $c_2 = 3$, $n_0 = 2$ abbiamo dimostrato che:

$$3n^2 - 2n - 1 \in \Theta(n^2)$$

Esempio 5.4 – Dimostrare che $5n + 3 \in O(n)$. Dobbiamo trovare $c > 0$ e $n_0 \geq 1$ tali che $5n + 3 \leq c \cdot n$ per ogni $n \geq n_0$.

Per $n \geq 1$: $\quad 5n + 3 \leq 5n + 3n = 8n$ (poiché $3 \leq 3n$ per $n \geq 1$).

Dunque con $c = 8$ e $n_0 = 1$ la disuguaglianza è verificata. Si ha $5n + 3 \in O(n)$.

Esempio 5.5 – Dimostrare che $n^2 \notin O(n)$. Per assurdo, supponiamo $n^2 \in O(n)$. Allora esistono $c, n_0 > 0$ tali che $n^2 \leq c \cdot n$ per ogni $n \geq n_0$, cioè $n \leq c$ per ogni $n \geq n_0$. Ma questo è impossibile, perché n cresce senza limite. Contraddizione.

Esempio 5.6 – Ordinamento per crescita asintotica. Ordinare le seguenti funzioni in ordine crescente di crescita asintotica:

$$2, \quad \log n, \quad (\log n)^2, \quad \sqrt{n}, \quad n, \quad n \log n, \quad n(\log n)^2, \quad n^2, \quad 2^n, \quad 3^n, \quad n!$$

Soluzione:

$$2 \prec \log n \prec (\log n)^2 \prec \sqrt{n} \prec n \prec n \log n \prec n(\log n)^2 \prec n^2 \prec 2^n \prec 3^n \prec n!$$

Dove $f \prec g$ significa $f(n) = o(g(n))$, ovvero $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Classificazione per famiglie di crescita:

- **Costanti**: $2 = \Theta(1)$
- **Logaritmiche**: $\log n$
- **Polilogaritmiche**: $(\log n)^2$
- **Sublineari**: $\sqrt{n} = n^{1/2}$
- **Lineari**: n
- **Linearitmiche**: $n \log n$, $n(\log n)^2$
- **Polinomiali**: n^2
- **Esponenziali**: $2^n \prec 3^n$ (base maggiore $\Rightarrow$ crescita più rapida)
- **Fattoriali**: $n!$ (cresce più rapidamente di qualsiasi c^n con c costante)

Nota. La notazione $\log^2 n$ si intende come $(\log n)^2$, non come $\log(\log n)$. Per evitare ambiguità, in queste dispense si usa sempre la scrittura estesa $(\log n)^2$.

5.2 Limiti inferiori alla difficoltà di un problema

La notazione asintotica ci consente di classificare la complessità dei singoli algoritmi. Un'ulteriore domanda fondamentale è: dato un problema, qual è il minimo costo necessario per risolverlo? Per rispondere, occorre stabilire dei **limiti inferiori**, cioè delle soglie al di sotto delle quali nessun algoritmo può scendere.

Definizione 5.8 – Difficoltà di un problema. Dato un problema π, la **difficoltà** di π è la complessità al caso pessimo del miglior algoritmo che risolve π, espressa in funzione della dimensione dell'input e in termini asintotici. Un algoritmo che risolve π con complessità $T(n)$ fornisce un **limite superiore** alla difficoltà di π.

Definizione 5.9 – Limite inferiore. Una funzione $L(n)$ è un **limite inferiore** per il problema π se, per ogni algoritmo A che risolve π, la complessità al caso pessimo di A soddisfa $T_{A(n)} \in \Omega(L(n))$.

In altre parole, $L(n)$ è il numero minimo di operazioni necessarie per risolvere π al caso peggiore.

Nota – Algoritmo ottimo. Un algoritmo è **ottimo** per il problema π se la sua complessità al caso pessimo coincide (asintoticamente) con il limite inferiore. In tal caso il limite inferiore è detto **stretto**.

Esistono tre criteri principali per stabilire limiti inferiori.

5.2.1 Criterio della dimensione dell'input

Se la soluzione di un problema richiede necessariamente l'esame di tutti i dati in input, allora la dimensione dell'input n è un limite inferiore:

$$L(n) = \Omega(n)$$

Esempio 5.7 – Ricerca del massimo. La ricerca del massimo in un vettore non ordinato deve necessariamente esaminare tutti gli n elementi: un elemento non esaminato potrebbe essere il massimo. Dunque $L(n) = \Omega(n)$.

Poiché l'algoritmo di scansione lineare ha complessità $\Theta(n)$, il limite inferiore è stretto e l'algoritmo è **ottimo**.

5.2.2 Criterio dell'albero di decisione

Questo criterio si applica a problemi risolvibili attraverso una sequenza di **decisioni binarie** (tipicamente confronti tra valori) che via via riducono lo spazio delle soluzioni possibili.

Definizione 5.10 – Albero di decisione. Un **albero di decisione** per un problema π è un albero binario in cui:

- ogni **nodo interno** rappresenta un confronto (decisione);
- ogni **foglia** rappresenta una possibile soluzione;
- ogni **percorso radice-foglia** corrisponde a una possibile esecuzione dell'algoritmo.

Il **caso pessimo** di un algoritmo basato su confronti corrisponde alla lunghezza del percorso più lungo dalla radice a una foglia, ossia all'**altezza** dell'albero di decisione.

Definizione 5.11 – Altezza di un albero. L'altezza di un albero è la lunghezza (in numero di archi) del più lungo percorso dalla radice ad una foglia.

Teorema 5.6 – Limite inferiore dall'albero di decisione. Se $\mathrm{SOL}(n)$ è il numero di soluzioni possibili per un'istanza di dimensione n del problema π, allora ogni albero di decisione per π ha almeno $\mathrm{SOL}(n)$ foglie. Poiché un albero binario con F foglie ha altezza almeno $\log_2 F$, si ha:

$$L(n) = \Omega(\log_2(\mathrm{SOL}(n)))$$

Dimostrazione. Un albero binario di altezza h ha al massimo 2^h foglie. L'albero di decisione deve avere almeno $\text{SOL}(n)$ foglie (una per ogni soluzione possibile). Dunque $2^h \geq \text{SOL}(n)$, da cui $h \geq \log_2(\text{SOL}(n))$. ■

Nota – Percorsi nell'albero di decisione. In un albero di decisione, il percorso più breve dalla radice a una foglia corrisponde al **caso ottimo**, mentre il percorso più lungo corrisponde al **caso pessimo**. Un **albero bilanciato** è un albero in cui il caso ottimo e il caso pessimo differiscono al più di una costante.

Definizione 5.12 – Albero bilanciato. Un albero si dice **bilanciato** se, per ogni nodo, le altezze dei suoi sottoalberi differiscono al più di una costante. Come conseguenza, un albero bilanciato con n nodi ha altezza $\Theta(\log n)$.

Nota – Algoritmo ottimo per problemi basati su confronti. L'algoritmo ottimo al caso pessimo è quello che minimizza l'altezza dell'albero di decisione. Questo corrisponde a un albero bilanciato, con altezza $\Theta(\log(\text{SOL}(n)))$.

5.2.3 Criterio degli eventi contabili

Se la soluzione di un problema richiede che un certo **evento** si ripeta un numero minimo di volte, e ogni occorrenza ha un certo costo, allora si ottiene un limite inferiore:

$$L(n) = (\text{numero minimo di ripetizioni dell'evento}) \times (\text{costo per evento})$$

Esempio 5.8 – Ordinamento per confronti. Nell'ordinamento per confronti, l'evento elementare è il **confronto** tra due elementi (costo $\Theta(1)$). Le soluzioni possibili sono le $n!$ permutazioni dell'array. Dall'albero di decisione:

$$L(n) = \Omega(\log_2(n!)) = \Omega(n \log n)$$

dove l'ultima uguaglianza segue dall'approssimazione di Stirling: $\log(n!) = \Theta(n \log n)$.

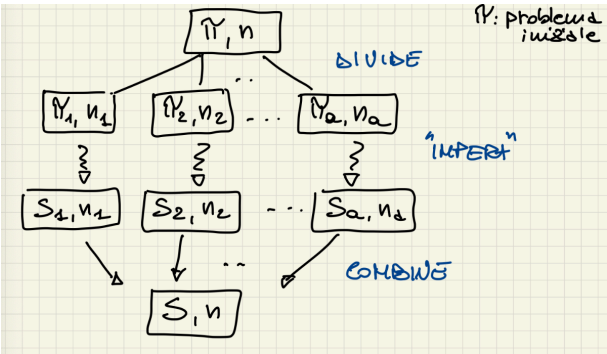
Poiché il Merge Sort ha complessità $\Theta(n \log n)$, esso è un algoritmo **ottimo** per l'ordinamento basato su confronti.

6 Divide et Impera

6.1 Il paradigma Divide et Impera

Il **Divide et Impera** (dal latino *divide et impera*) e uno dei paradigmi algoritmici fondamentali. Alla base della sua struttura vi sono tre fasi:

1. **Divide**: si suddivide il problema originale in due o più sotto-problemi *dello stesso tipo*, ciascuno operante su un sottoinsieme dei dati originali. La dimensione di ciascun sotto-problema e strettamente minore di quella del problema di partenza.
2. **Impera** (conquista): si risolvono i sotto-problemi in modo ricorsivo, applicando la stessa tecnica. Quando la dimensione del sotto-problema raggiunge un caso base (dimensione sufficientemente piccola), lo si risolve direttamente.
3. **Combina**: si fondono le soluzioni dei sotto-problemi per costruire la soluzione del problema originale.



La struttura generale di un algoritmo Divide et Impera e la seguente:

Schema generale Divide et Impera

```
soluzione DivideEtImpera(problema P) {
    if (P e un caso base) {
        return risolviDirettamente(P);
    }
    dividi P in sotto-problemi P1, P2, ..., Pa;
    soluzione s1 = DivideEtImpera(P1);
    soluzione s2 = DivideEtImpera(P2);
    ...
    soluzione sa = DivideEtImpera(Pa);
    return combina(s1, s2, ..., sa);
}
```

La complessità di un algoritmo Divide et Impera e descritta da una **relazione di ricorrenza** della forma:

$$T(n) = \underbrace{a}_{\text{num. sotto-problemi}} \cdot T(n/b) + \underbrace{f(n)}_{\text{costo divide + combina}}$$

dove $a \geq 1$ e il numero di sotto-problemi, $b > 1$ e il fattore di riduzione, e $f(n)$ rappresenta il costo delle fasi di divisione e combinazione.

6.2 Ricerca Binaria

La ricerca binaria e un algoritmo classico di tipo Divide et Impera per cercare un elemento k in un array $A[p..r]$ **ordinato**. L'idea e la seguente: si confronta k con l'elemento centrale dell'array; se k e uguale all'elemento centrale, la ricerca termina; altrimenti si prosegue ricorsivamente nella meta sinistra (se k e minore) o nella meta destra (se k e maggiore).

6.2.1 Implementazione e complessità

Input: array ordinato $A[p..r]$ di interi, elemento k da cercare.
Output: indice i tale che $A[i] = k$, oppure -1 se k non e presente.

Ricerca Binaria

```
int binarySearch(int[] A, int p, int r, int k){
    if(p > r){
        return -1;
    }
    if(p == r){
        if(A[p] == k){
            return p;
        } else {
            return -1;
        }
    }
    int q = (p + r) / 2;
    if(A[q] == k){
        return q;
    }
    if(A[q] > k){
        return binarySearch(A, p, q - 1, k);
    } else {
        return binarySearch(A, q + 1, r, k);
    }
}
```

Analizziamo la complessità. Sia $n = r - p + 1$ la dimensione del sotto-array corrente:

- **Divide**: il calcolo del punto medio q costa $\Theta(1)$.
- **Impera**: si effettua **una sola** chiamata ricorsiva su un sotto-array di dimensione al più$n/2$.
- **Combina**: il risultato della chiamata ricorsiva e gia la risposta, quindi il costo di combinazione e $\Theta(1)$.

La relazione di ricorrenza e:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(n/2) + \Theta(1) & \text{se } n \geq 2 \end{cases}$$

dove $f(n) = D(n) + C(n) = \Theta(1)$ (costo di divide + combina).

> **Teorema 6.1 – Complessita della Ricerca Binaria.** La ricerca binaria su un array ordinato di n elementi ha complessità nel caso pessimo $\Theta(\log n)$ e nel caso ottimo $\Theta(1)$ (quando l'elemento cercato si trova esattamente nella posizione centrale al primo confronto).

6.2.2 Correttezza

La correttezza di un algoritmo Divide et Impera si dimostra tipicamente per **induzione forte** sulla dimensione n del problema.

- **Caso base** ($n \leq 1$): se $p > r$ l'array e vuoto e si restituisce -1; se $p = r$ si confronta direttamente $A[p]$ con k. In entrambi i casi l'algoritmo e corretto.
- **Passo induttivo**: si assume, per **ipotesi induttiva**, che l'algoritmo sia corretto per ogni input di dimensione $n' < n$ (con $n' \geq 0$). Si deve dimostrare che e corretto per input di dimensione n. Poiché l'array e ordinato, dopo il confronto con $A[q]$:
  - se $A[q] = k$, l'indice q e restituito correttamente;
  - se $A[q] > k$, allora k puo trovarsi solo in $A[p..q-1]$ (dimensione $< n$), e per ipotesi induttiva la chiamata ricorsiva restituisce il risultato corretto;
  - se $A[q] < k$, il ragionamento e simmetrico su $A[q+1..r]$.

6.2.3 Varianti della Ricerca Binaria

La ricerca binaria standard trova *una* occorrenza di un elemento, ma non garantisce quale (prima, ultima, o una qualsiasi). Le seguenti varianti permettono di trovare specificamente la prima o l'ultima occorrenza.

6.2.3.1 Ricerca Binaria Sinistra

Trova la **prima occorrenza** (più asinistra) di un elemento k in un array ordinato.

Ricerca Binaria Sinistra

```
int ricercaBinariaSx(int[] a, int sx, int dx, int k) {
    if (sx > dx) {
        return -1;
    }
    int cx = (sx + dx) / 2;
    if (a[cx] == k and (cx == sx or a[cx - 1] != k)) {
        return cx;
    }
    if (a[cx] >= k) {
        return ricercaBinariaSx(a, sx, cx - 1, k);
    } else {
        return ricercaBinariaSx(a, cx + 1, dx, k);
    }
}
```

Idea: quando troviamo k in posizione cx, verifichiamo se e la prima occorrenza controllando che:
- $\text{cx} = \text{sx}$ (siamo al bordo sinistro del sotto-array), oppure
- $a[\text{cx} - 1] \neq k$ (l'elemento precedente e diverso da k).

Se la condizione non e soddisfatta, k compare anche a sinistra di cx, quindi si prosegue la ricerca nella meta sinistra. La complessità resta $O(\log n)$.

6.2.3.2 Ricerca Binaria Destra

Trova l'**ultima occorrenza** (più adestra) di un elemento k in un array ordinato.

Ricerca Binaria Destra

```
int ricercaBinariaDx(int[] a, int sx, int dx, int k) {
    if (sx > dx) {
        return -1;
    }
    int cx = (sx + dx) / 2;
    if (a[cx] == k and (cx == dx or a[cx + 1] != k)) {
        return cx;
    }
```

```
    if (a[cx] <= k) {
        return ricercaBinariaDx(a, cx + 1, dx, k);
    } else {
        return ricercaBinariaDx(a, sx, cx - 1, k);
    }
}
```

Idea: simmetricamente alla variante sinistra, quando troviamo k in posizione cx, verifichiamo se e l'ultima occorrenza controllando che:

- $\text{cx} = \text{dx}$ (siamo al bordo destro del sotto-array), oppure
- $a[\text{cx}+1] \neq k$ (l'elemento successivo e diverso da k).

Se la condizione non e soddisfatta, si prosegue nella meta destra. La complessità resta $O(\log n)$.

6.2.3.3 Conta Occorrenze

Combinando le due varianti si puo contare il numero di occorrenze di k in un array ordinato in tempo $\Theta(\log n)$.

Conta Occorrenze

```
int contaOccorrenze(int[] a, int n, int k) {
    int prima = ricercaBinariaSx(a, 0, n - 1, k);
    if (prima == -1) {
        return 0;
    }
    int ultima = ricercaBinariaDx(a, 0, n - 1, k);
    return ultima - prima + 1;
}
```

> **Nota – Complessita di contaOccorrenze.** Si eseguono al massimo due ricerche binarie, ciascuna con complessità $O(\log n)$: la complessità totale e quindi $\Theta(\log n)$. Un approccio lineare che scorre l'intero array richiederebbe $\Theta(n)$, dunque le varianti della ricerca binaria offrono un miglioramento significativo per array di grandi dimensioni.

6.3 Merge Sort

Il Merge Sort e un algoritmo di ordinamento basato sul paradigma Divide et Impera. L'idea e la seguente:

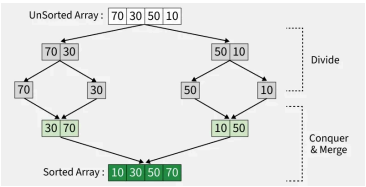
1. **Divide**: si divide l'array $A[p..r]$ in due meta $A[p..q]$ e $A[q+1..r]$, dove $q = \lfloor (p+r)/2 \rfloor$.
2. **Impera**: si ordinano ricorsivamente le due meta.
3. **Combina**: si fondono (*merge*) le due meta ordinate in un unico array ordinato.

6.3.1 Implementazione

Merge Sort

```
mergeSort(int[] A, int p, int r){        // -- T(n)
    if(p < r){                           // -- Theta(1)
        int q = (p + r) / 2;             // divide -- Theta(1)
        mergeSort(A, p, q);              // impera -- T(n/2)
        mergeSort(A, q + 1, r);          // impera -- T(n/2)
        merge(A, p, q, r);               // combina -- Theta(n)
```

```
    }
}
```



La procedura `merge` fonde due sotto-array ordinati $A[p..q]$ e $A[q+1..r]$ in un unico sotto-array ordinato $A[p..r]$, utilizzando due array ausiliari L e R.

Procedura Merge

```
merge(int[] A, int p, int q, int r){
    int n1 = q - p + 1;
    int n2 = r - q;
    int[] L = new int[n1 + 1];
    int[] R = new int[n2 + 1];

    int i = 1;
    while(i <= n1){
        L[i] := A[p + i - 1];
        i := i + 1;
    }
    int j = 1;
    while(j <= n2){
        R[j] := A[q + j];
        j := j + 1;
    }
    L[n1 + 1] := +∞;
    R[n2 + 1] := +∞;

    i := 1;
    j := 1;
    int k = p;
    while(k <= r){
        if(L[i] <= R[j]){
            A[k] := L[i];
            i := i + 1;
        } else {
            A[k] := R[j];
            j := j + 1;
        }
        k := k + 1;
    }
}
```

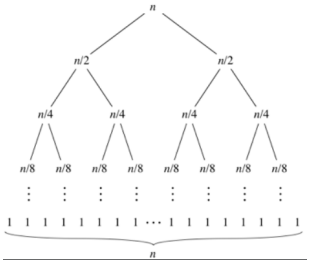
> **Nota – Funzionamento di Merge.** La procedura `merge` utilizza due **sentinelle** $+\infty$ alla fine degli array ausiliari L e R: quando un array ausiliario e stato completamente percorso, la sentinella garantisce che il confronto selezioni sempre l'elemento dall'altro array, senza necessità di controlli aggiuntivi sugli indici. Ogni iterazione del ciclo `while` copia esattamente un elemento in A, per un totale di $n_1 + n_2 = r - p + 1$ iterazioni. La complessità di `merge` e dunque $\Theta(n)$.

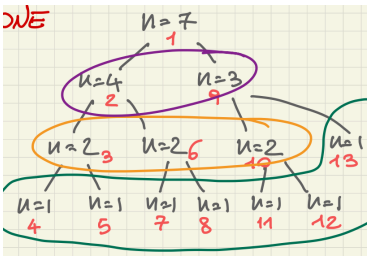
6.3.2 Relazione di ricorrenza e complessità

La relazione di ricorrenza del Merge Sort e:

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

Per comprendere intuitivamente la complessità, si puo analizzare l'**albero di ricorsione**: a ciascun livello i dell'albero vi sono 2^i sotto-problemi, ciascuno di dimensione $n/2^i$. Il costo al livello i e $2^i \cdot c \cdot (n/2^i) = c \cdot n = \Theta(n)$. L'albero ha $\log_2 n$ livelli, dunque il costo totale e $\Theta(n \log n)$.





> **Teorema 6.2 – Complessita del Merge Sort.** La complessità in tempo del Merge Sort e $\Theta(n \log n)$ in **tutti i casi** (ottimo, medio, pessimo). La complessità in spazio e $O(n)$, dovuta agli array ausiliari utilizzati dalla procedura `merge`.

> **Nota – Relazione di ricorrenza vs. complessità.** La **relazione di ricorrenza** e la definizione matematica del costo di un algoritmo ricorsivo in funzione dell'input: descrive $T(n)$ in termini di T applicata a sotto-problemi piùpiccoli. La **complessità** e il risultato della risoluzione di tale relazione, espressa in notazione asintotica.

6.4 Relazioni di ricorrenza

Le relazioni di ricorrenza sono lo strumento matematico per descrivere la complessità $T(n)$ di algoritmi ricorsivi. Distinguiamo due forme principali.

6.4.1 Relazioni bilanciate

Una relazione di ricorrenza si dice **bilanciata** quando i sotto-problemi hanno tutti la stessa dimensione:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq n_0 \\ \underbrace{a}_{\text{sotto-problemi}} \cdot T(n/b) + \underbrace{f(n)}_{\text{forzante}} & \text{se } n > n_0 \end{cases}$$

dove $a \in \mathbb{N}^+$ (numero di sotto-problemi), $b > 1, b \in \mathbb{Q}$ (fattore di riduzione), e $f(n)$ e il termine **forzante** che rappresenta il costo delle fasi di divisione e combinazione.

La ricerca binaria ($a = 1$, $b = 2$, $f(n) = \Theta(1)$) e il Merge Sort ($a = 2$, $b = 2$, $f(n) = \Theta(n)$) sono esempi di relazioni bilanciate.

6.4.2 Relazioni di ordine k

Una relazione e di **ordine k** quando $T(n)$ dipende dai k valori precedenti:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq n_0 \\ \alpha_1 T(n-1) + \alpha_2 T(n-2) + \cdots + \alpha_k T(n-k) + f(n) & \text{se } n > n_0 \end{cases}$$

Queste relazioni si risolvono con il **metodo dell'equazione caratteristica**. L'idea e la seguente: per la parte omogenea (cioè con $f(n) = 0$), si cerca una soluzione della forma $T(n) = x^n$. Sostituendo nella ricorrenza si ottiene:

$$x^n = \alpha_1 x^{n-1} + \alpha_2 x^{n-2} + \cdots + \alpha_k x^{n-k}$$

Dividendo per x^{n-k} si ricava l'**equazione caratteristica**:

$$x^k - \alpha_1 x^{k-1} - \alpha_2 x^{k-2} - \cdots - \alpha_k = 0$$

Se le k radici $x_1, x_2, ..., x_k$ sono distinte, la soluzione generale e:

$$T(n) = c_1 x_1^n + c_2 x_2^n + \cdots + c_k x_k^n$$

dove le costanti $c_1, ..., c_k$ si determinano dalle condizioni iniziali. Dal punto di vista asintotico, il termine dominante e quello con la radice di modulo massimo.

> **Esempio 6.1 – Fibonacci come relazione di ordine 2.** La successione di Fibonacci soddisfa $T(n) = T(n-1) + T(n-2)$, con $\alpha_1 = 1$ e $\alpha_2 = 1$.
>
> L'equazione caratteristica e:
>
> $$x^2 - x - 1 = 0$$
>
> Le radici sono:
>
> $$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618 \qquad \hat{\varphi} = \frac{1-\sqrt{5}}{2} \approx -0.618$$
>
> La soluzione generale e $T(n) = c_1 \varphi^n + c_2 \hat{\varphi}^n$. Poiché $|\hat{\varphi}| < 1$, il termine $\hat{\varphi}^n \to 0$ e il termine dominante e φ^n, da cui:
>
> $$T(n) = \Theta(\varphi^n) \approx \Theta(1.618^n)$$
>
> La complessità della successione di Fibonacci e dunque **esponenziale**.

6.4.3 Metodi di risoluzione

Esistono quattro metodi principali per risolvere le relazioni di ricorrenza:

1. **Metodo iterativo**: si espande (srotola) la ricorrenza fino a raggiungere i casi base, poi si somma il lavoro a tutti i livelli.
2. **Metodo di sostituzione**: si ipotizza una soluzione e la si dimostra per induzione.
3. **Albero di ricorsione**: ausilio grafico che rappresenta i costi ai vari livelli della ricorsione; spesso usato per formulare un'ipotesi da verificare col metodo di sostituzione.
4. **Master Theorem**: formula chiusa applicabile alle relazioni bilanciate (vedi sezione successiva).

Esempio 6.2 – Metodo iterativo: MergeSort. Consideriamo la ricorrenza del MergeSort: $T(n) = 2T(n/2) + n$ (con $T(1) = \Theta(1)$).

Si **srotola** (unrolling) la ricorrenza sostituendo ripetutamente la definizione di T:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n = 8T(n/8) + 3n \\ &= \cdots \\ &= 2^k T\big(n/2^k\big) + kn \end{aligned}$$

Il processo termina quando il sotto-problema raggiunge il caso base, cioè quando $n/2^k = 1$, ovvero $k = \log_2 n$. Sostituendo:

$$T(n) = 2^{\log_2 n} \cdot T(1) + n \log_2 n = n \cdot \Theta(1) + n \log n = \Theta(n \log n)$$

Il risultato coincide con quello ottenuto tramite il Master Theorem.

6.5 Master Theorem

Il Master Theorem fornisce una soluzione in forma chiusa per le relazioni di ricorrenza bilanciate.

Teorema 6.3 – Master Theorem. Sia data la relazione di ricorrenza:

$$T(n) = aT(n/b) + f(n)$$

con $a \geq 1$, $b > 1$ e $f(n) > 0$ definitivamente. Sia $c_{\text{crit}} = \log_b a$. Allora:

1. **Caso 1** – $f(n)$ cresce piùlentamente di $n^{c_{\text{crit}}}$:
   Se $\exists \varepsilon > 0 : f(n) = O\big(n^{\log_b a - \varepsilon}\big)$, allora $T(n) = \Theta\big(n^{\log_b a}\big)$.
2. **Caso 2** – $f(n)$ cresce come $n^{c_{\text{crit}}}$ (a meno di fattori logaritmici):
   Se $f(n) = \Theta\big(n^{\log_b a}\big)$, allora $T(n) = \Theta\big(n^{\log_b a} \cdot \log n\big)$.
3. **Caso 3** – $f(n)$ cresce piùvelocemente di $n^{c_{\text{crit}}}$:
   Se $\exists \varepsilon > 0 : f(n) = \Omega\big(n^{\log_b a + \varepsilon}\big)$ e inoltre $\exists c < 1 : a \cdot f(n/b) \leq c \cdot f(n)$ (condizione di regolarità), allora $T(n) = \Theta(f(n))$.

Nota – Caso 2 generalizzato. Il secondo caso ammette una formulazione piùgenerale:

$$\exists k \geq 0 : f(n) = \Theta\big(n^{\log_b a} \cdot \log^k n\big) \Rightarrow T(n) = \Theta\big(n^{\log_b a} \cdot \log^{k+1} n\big)$$

Il caso standard corrisponde a $k = 0$.

Nota – Interpretazione intuitiva. Il Master Theorem confronta il lavoro svolto dalla forzante $f(n)$ con il costo intrinseco della ricorsione, misurato da $n^{\log_b a}$:

- **Caso 1**: la ricorsione domina: il costo totale e determinato dal numero di foglie dell'albero di ricorsione.
- **Caso 2**: forzante e ricorsione contribuiscono in modo bilanciato: il costo e amplificato da un fattore $\log n$ (uno per livello dell'albero).
- **Caso 3**: la forzante domina: il costo totale e determinato dal lavoro alla radice dell'albero.

6.5.1 Come applicare il Master Theorem

Nota – Procedimento.
1. Identificare i parametri a, b e $f(n)$.
2. Calcolare l'**esponente critico** $c_{\text{crit}} = \log_b a$.
3. Confrontare la crescita di $f(n)$ con $n^{c_{\text{crit}}}$:
   - se $f(n)$ cresce **polinomialmente piùlentamente** di $n^{c_{\text{crit}}} \Rightarrow$ Caso 1;
   - se $f(n)$ cresce **allo stesso modo** (a meno di fattori logaritmici) $\Rightarrow$ Caso 2;
   - se $f(n)$ cresce **polinomialmente piùvelocemente** di $n^{c_{\text{crit}}} \Rightarrow$ Caso 3 (verificare la condizione di regolarità).

6.5.2 Esempi di applicazione

Esempio 6.3 – Caso 2: Ricerca Binaria. $T(n) = T(n/2) + \Theta(1)$

Parametri: $a = 1$, $b = 2$, $f(n) = \Theta(1)$.

Esponente critico: $c_{\text{crit}} = \log_2 1 = 0$, dunque $n^{c_{\text{crit}}} = n^0 = 1$.

Confronto: $f(n) = \Theta(1) = \Theta(n^0) = \Theta\big(n^{\log_b a}\big)$.

$\Rightarrow$ **Caso 2**: $T(n) = \Theta(n^0 \cdot \log n) = \Theta(\log n)$.

Esempio 6.4 – Caso 2: Merge Sort. $T(n) = 2T(n/2) + \Theta(n)$

Parametri: $a = 2$, $b = 2$, $f(n) = \Theta(n)$.

Esponente critico: $c_{\text{crit}} = \log_2 2 = 1$, dunque $n^{c_{\text{crit}}} = n$.

Confronto: $f(n) = \Theta(n) = \Theta\big(n^{\log_b a}\big)$.

$\Rightarrow$ **Caso 2**: $T(n) = \Theta(n \cdot \log n)$.

Esempio 6.5 – Caso 1: Algoritmo ipotetico. $T(n) = 4T(n/2) + n$

Parametri: $a = 4$, $b = 2$, $f(n) = n$.

Esponente critico: $c_{\text{crit}} = \log_2 4 = 2$, dunque $n^{c_{\text{crit}}} = n^2$.

Confronto: $f(n) = n = O(n^{2-\varepsilon})$ con $\varepsilon = 1$. La forzante cresce piùlentamente di n^2.

$\Rightarrow$ **Caso 1**: $T(n) = \Theta(n^2)$.

Interpretazione: il costo e dominato dalle $4^{\log_2 n} = n^2$ foglie dell'albero di ricorsione.

Esempio 6.6 – Caso 3: Algoritmo ipotetico. $T(n) = T(n/2) + n$

Parametri: $a = 1$, $b = 2$, $f(n) = n$.

Esponente critico: $c_{\text{crit}} = \log_2 1 = 0$, dunque $n^{c_{\text{crit}}} = 1$.

Confronto: $f(n) = n = \Omega(n^{0+\varepsilon})$ con $\varepsilon = 1$. La forzante cresce piùvelocemente di $n^0 = 1$.

Verifica condizione di regolarità: $a \cdot f(n/b) = 1 \cdot n/2 = n/2 \leq c \cdot n$ con $c = 1/2 < 1$. ✓

$\Rightarrow$ **Caso 3**: $T(n) = \Theta(n)$.

Interpretazione: il costo e dominato dal lavoro alla radice.

Esempio 6.7 – Caso non coperto dal Master Theorem. $T(n) = 2T(n/2) + n \log n$

Parametri: $a = 2$, $b = 2$, $f(n) = n \log n$.

Esponente critico: $c_{\text{crit}} = \log_2 2 = 1$, dunque $n^{c_{\text{crit}}} = n$.

Si ha $f(n) = n \log n$, che cresce più di n ma **non polinomialmente** più di n (non esiste $\varepsilon > 0$ tale che $n \log n = \Omega(n^{1+\varepsilon})$). Non si applica il Caso 3.

Tuttavia, applicando il **Caso 2 generalizzato** con $k = 1$: $f(n) = \Theta(n \cdot \log^1 n)$, si ottiene $T(n) = \Theta(n \cdot \log^2 n)$.

7 Algoritmi di Ordinamento

7.1 Insertion Sort

L'Insertion Sort è un algoritmo di ordinamento iterativo. L'idea è quella di mantenere una porzione iniziale dell'array già ordinata e, ad ogni passo, inserire il prossimo elemento nella posizione corretta all'interno di tale porzione.

Input: array $A[1..n]$ di interi.
Output: A ordinato in modo non decrescente: $A[1] \leq A[2] \leq ... \leq A[n]$.

InsertionSort

```
insertionSort(int[] A, int n){
    int j = 2;
    while(j <= n){
        int k = A[j];
        int i = j - 1;
        while((i > 0) && (A[i] > k)){
            A[i + 1] := A[i];
            i := i - 1;
        }
        A[i + 1] := k;
        j := j + 1;
    }
}
```

Funzionamento. All'iterazione j-esima del ciclo esterno, l'elemento $A[j]$ viene salvato in una variabile k (la *chiave*). Il ciclo interno scorre il sottoarray $A[1..j-1]$ da destra verso sinistra, spostando a destra tutti gli elementi maggiori di k. Quando si trova la posizione corretta (un elemento minore o uguale a k, oppure l'inizio dell'array), la chiave viene inserita.

Esempio 7.1 – Esecuzione di Insertion Sort. Sia $A = [5, 2, 4, 6, 1, 3]$ con $n = 6$.

- $\boldsymbol{j = 2}$: chiave $k = 2$. Confronto con $A[1] = 5 > 2$: sposto 5 a destra. Raggiunto l'inizio dell'array: inserisco k.
  $A = [2, 5, 4, 6, 1, 3]$
- $\boldsymbol{j = 3}$: chiave $k = 4$. Confronto con $A[2] = 5 > 4$: sposto 5. Confronto con $A[1] = 2 \leq 4$: stop. Inserisco k.
  $A = [2, 4, 5, 6, 1, 3]$
- $\boldsymbol{j = 4}$: chiave $k = 6$. Confronto con $A[3] = 5 \leq 6$: stop. Nessuno spostamento.
  $A = [2, 4, 5, 6, 1, 3]$
- $\boldsymbol{j = 5}$: chiave $k = 1$. Sposto $6, 5, 4, 2$ (tutti maggiori di 1). Raggiunto l'inizio: inserisco k.
  $A = [1, 2, 4, 5, 6, 3]$
- $\boldsymbol{j = 6}$: chiave $k = 3$. Sposto $6, 5, 4$. Confronto con $A[2] = 2 \leq 3$: stop. Inserisco k.
  $A = [1, 2, 3, 4, 5, 6]$

7.1.1 Dimostrazione di correttezza con invariante di ciclo

Per dimostrare che l'Insertion Sort produce effettivamente un array ordinato, utilizziamo la tecnica dell'**invariante di ciclo**.

Invariante: All'inizio dell'iterazione j-esima del ciclo esterno, il sottoarray $A[1..j-1]$ contiene gli stessi elementi che erano in $A[1..j-1]$ prima dell'esecuzione dell'algoritmo, disposti in ordine non decrescente.

Dimostrazione. **Inizializzazione** ($j = 2$):
Prima della prima iterazione, il sottoarray $A[1..1]$ contiene un solo elemento. Un singolo elemento è banalmente ordinato, e coincide con l'elemento originale. L'invariante è verificato.

Mantenimento: Supponiamo che l'invariante sia vero all'inizio dell'iterazione j-esima, ovvero $A[1..j-1]$ è ordinato e contiene gli elementi originali. Il corpo del ciclo:
1. Salva $A[j]$ nella variabile k.
2. Il ciclo interno sposta a destra gli elementi di $A[1..j-1]$ che sono maggiori di k, preservando il loro ordine relativo.
3. Inserisce k nella posizione corretta, cioè immediatamente dopo l'ultimo elemento $\leq k$.

Al termine, $A[1..j]$ contiene tutti gli elementi originali di $A[1..j]$ ed è ordinato. L'invariante vale quindi per $j+1$.

Terminazione ($j = n+1$):
Il ciclo termina quando $j = n+1$. Per l'invariante, $A[1..n]$ contiene gli elementi originali in ordine non decrescente. Questo è esattamente la specifica dell'output desiderato. ■

7.1.2 Analisi della complessità

Sia d_j il numero di volte in cui il ciclo interno viene eseguito (con guardia vera) per un dato valore di j. Al variare di j da 2 a n, il numero totale di confronti è:

$$C(n) = \sum_{j=2}^{n} d_j$$

- **Caso ottimo**: l'array è già ordinato. Per ogni j, la condizione $A[i] > k$ è subito falsa, dunque $d_j = 0$ spostamenti vengono effettuati. Tuttavia, anche quando $d_j = 0$, si esegue comunque un confronto: il test di guardia del ciclo interno ($A[i] > k$) che risulta immediatamente falso. Per questo il costo per ogni j è di 1 confronto. Il numero totale di confronti è:

  $$C(n) = \sum_{j=2}^{n} 1 = n - 1 = \Theta(n)$$

  La complessità è **lineare**.

- **Caso pessimo**: l'array è ordinato in ordine decrescente. Per ogni j, tutti gli elementi di $A[1..j-1]$ sono maggiori della chiave, e il ciclo interno esegue $j-1$ spostamenti. Il numero totale di confronti è:

  $$C(n) = \sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$$

  La complessità è **quadratica**.

In sintesi:
- **Caso ottimo**: $T(n) = \Theta(n)$ – lineare
- **Caso pessimo**: $T(n) = \Theta(n^2)$ – quadratico

L'Insertion Sort è dunque un algoritmo efficiente su array quasi ordinati, ma inadeguato per array in ordine inverso o casuale di grandi dimensioni.

7.2 Selection Sort

Il Selection Sort è un algoritmo di ordinamento iterativo basato su un'idea diversa dall'Insertion Sort: ad ogni passo, si seleziona il minimo tra gli elementi non ancora ordinati e lo si colloca nella posizione corretta tramite uno scambio.

Input: array $A[1..n]$ di interi.
Output: A ordinato in modo non decrescente: $A[1] \leq A[2] \leq ... \leq A[n]$.

SelectionSort

```
selectionSort(int[] A, int n){
    int i = 1;
    while(i <= n - 1){
        int min = i;
        int j = i + 1;
        while(j <= n){
            if(A[j] < A[min]){
                min := j;
            }
            j := j + 1;
        }
        // swap(A[i], A[min])
        int temp = A[i];
        A[i] := A[min];
        A[min] := temp;
        i := i + 1;
    }
}
```

Funzionamento. All'iterazione i-esima del ciclo esterno, il ciclo interno cerca l'indice dell'elemento minimo nel sottoarray $A[i..n]$. Una volta trovato, l'elemento minimo viene scambiato con $A[i]$. Dopo questa operazione, $A[1..i]$ contiene gli i elementi più piccoli dell'array, in ordine non decrescente.

Esempio 7.2 – Esecuzione di Selection Sort. Sia $A = [5, 2, 4, 6, 1, 3]$ con $n = 6$.

- $\boldsymbol{i = 1}$: cerco il minimo in $A[1..6]$. Minimo: $A[5] = 1$. Scambio $A[1]$ con $A[5]$.
  $A = [1, 2, 4, 6, 5, 3]$
- $\boldsymbol{i = 2}$: cerco il minimo in $A[2..6]$. Minimo: $A[2] = 2$. Scambio $A[2]$ con se stesso.
  $A = [1, 2, 4, 6, 5, 3]$
- $\boldsymbol{i = 3}$: cerco il minimo in $A[3..6]$. Minimo: $A[6] = 3$. Scambio $A[3]$ con $A[6]$.
  $A = [1, 2, 3, 6, 5, 4]$
- $\boldsymbol{i = 4}$: cerco il minimo in $A[4..6]$. Minimo: $A[6] = 4$. Scambio $A[4]$ con $A[6]$.
  $A = [1, 2, 3, 4, 5, 6]$
- $\boldsymbol{i = 5}$: cerco il minimo in $A[5..6]$. Minimo: $A[5] = 5$. Scambio $A[5]$ con se stesso.
  $A = [1, 2, 3, 4, 5, 6]$

Nota – Invariante del ciclo interno del Selection Sort. Il ciclo interno del Selection Sort mantiene la seguente invariante: all'inizio dell'iterazione j-esima del ciclo interno, $A[\text{min}]$ è il minimo di $A[i..j-1]$. Questo garantisce che, al termine del ciclo interno (quando $j = n + 1$), $A[\text{min}]$ sia effettivamente il minimo dell'intero sottoarray $A[i..n]$.

7.2.1 Dimostrazione di correttezza con invariante di ciclo

Invariante: All'inizio dell'iterazione i-esima del ciclo esterno:

1. Il sottoarray $A[1..i-1]$ è ordinato in modo non decrescente.
2. Ogni elemento di $A[1..i-1]$ è minore o uguale ad ogni elemento di $A[i..n]$.

Dimostrazione. **Inizializzazione** ($i = 1$):
Il sottoarray $A[1..0]$ è vuoto. Entrambe le condizioni dell'invariante sono banalmente verificate (proprietà dell'insieme vuoto).

Mantenimento: Supponiamo l'invariante vero per i. Dimostriamo che vale per $i+1$.

- Il ciclo interno esamina $A[i], A[i+1], ..., A[n]$ e individua l'indice min dell'elemento minimo in questo sottoarray.
- Lo scambio tra $A[i]$ e $A[\text{min}]$ colloca in posizione i il più piccolo elemento di $A[i..n]$.
- Per l'ipotesi induttiva (condizione 2), tutti gli elementi di $A[1..i-1]$ sono $\leq$ ad ogni elemento di $A[i..n]$, e in particolare $\leq A[i]$ dopo lo scambio.
- Dunque $A[1..i]$ è ordinato e contiene gli i elementi più piccoli dell'array.
- Ogni elemento di $A[i+1..n]$ è $\geq A[i]$ (perché $A[i]$ era il minimo di $A[i..n]$).

L'invariante vale per $i+1$.

Terminazione ($i = n$):
Per l'invariante, $A[1..n-1]$ è ordinato e contiene gli $n-1$ elementi più piccoli. L'unico elemento rimasto, $A[n]$, è necessariamente il più grande. L'intero array è ordinato. ■

7.2.2 Analisi della complessità

Il ciclo interno, per un dato valore di i, esegue esattamente $n - i$ confronti. Il numero totale di confronti è:

$$C(n) = \sum_{i=1}^{n-1}(n-i) = (n-1) + (n-2) + ... + 1 = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$$

A differenza dell'Insertion Sort, il Selection Sort esegue sempre $\Theta(n^2)$ confronti, **indipendentemente dall'input**. Il caso ottimo, pessimo e medio coincidono:

$$T(n) = \Theta(n^2)$$

> **Nota – Confronto tra Insertion Sort e Selection Sort.**
> - L'Insertion Sort ha complessità $\Theta(n)$ nel caso ottimo (array ordinato) e $\Theta(n^2)$ nel caso pessimo.
> - Il Selection Sort ha complessità $\Theta(n^2)$ in tutti i casi.
> - L'Insertion Sort è quindi preferibile quando l'input può essere parzialmente ordinato.
> - Il Selection Sort ha il vantaggio di eseguire al più $n-1$ scambi (utile quando le operazioni di scrittura sono costose).

> **Nota – Stabilità degli algoritmi di ordinamento.** Un algoritmo di ordinamento si dice **stabile** se preserva l'ordine relativo degli elementi con la stessa chiave.
>
> - **Insertion Sort è stabile**: il ciclo interno si arresta quando incontra un elemento $\leq k$ (la condizione è $A[i] > k$, strettamente maggiore). Pertanto, elementi uguali alla chiave non vengono spostati e mantengono la loro posizione relativa originale.
> - **Selection Sort non è stabile**: lo scambio tra $A[i]$ e $A[\text{min}]$ può alterare l'ordine relativo di elementi uguali. Ad esempio, con $A = [2_a, 2_b, 1]$, alla prima iterazione si scambia 2_a con 1, ottenendo $[1, 2_b, 2_a]$: l'ordine relativo di 2_a e 2_b è invertito.

7.3 Invariante di ciclo: schema generale

La tecnica dell'invariante di ciclo è uno strumento fondamentale per dimostrare la **correttezza** degli algoritmi iterativi. Formalizziamo la sua struttura.

> **Definizione 7.1 – Invariante di ciclo.** Un **invariante di ciclo** è una proprietà logica P tale che:
> 1. **Inizializzazione**: P è vera prima della prima iterazione del ciclo.
> 2. **Mantenimento**: se P è vera all'inizio di un'iterazione, allora è vera anche all'inizio dell'iterazione successiva.
> 3. **Terminazione**: quando il ciclo termina, l'invariante P, combinata con la condizione di uscita dal ciclo, implica la correttezza dell'algoritmo.

> **Nota – Analogia con l'induzione matematica.** La struttura della dimostrazione con invariante di ciclo ricalca il **principio di induzione**:
> - L'inizializzazione corrisponde al **caso base**.
> - Il mantenimento corrisponde al **passo induttivo**.
> - La terminazione sfrutta il fatto che il ciclo ha un numero finito di iterazioni (analogamente alla buona fondatezza dell'induzione sui naturali).

> **Nota – Schema pratico per le dimostrazioni.** Per dimostrare la correttezza di un algoritmo iterativo con invariante di ciclo:
> 1. **Identificare l'invariante**: determinare quale proprietà rimane vera ad ogni iterazione.
> 2. **Inizializzazione**: verificare che l'invariante valga prima della prima iterazione.
> 3. **Mantenimento**: assumere che l'invariante valga all'iterazione k e dimostrare che vale per $k+1$.
> 4. **Terminazione**: combinare l'invariante con la condizione di uscita dal ciclo per dedurre la correttezza del risultato.

7.4 QuickSort

Il **QuickSort** è un algoritmo di ordinamento basato sul paradigma Divide et Impera, inventato da Tony Hoare nel 1960. È uno degli algoritmi di ordinamento più utilizzati nella pratica grazie alla sua eccellente efficienza nel caso medio e al basso overhead di memoria, poiché opera **in loco** (in-place) senza richiedere array ausiliari.

L'idea chiave consiste nello scegliere un elemento detto **pivot** e nel partizionare l'array in due sottoinsiemi:

- tutti gli elementi $\leq$ pivot finiscono nella parte sinistra
- tutti gli elementi $>$ pivot finiscono nella parte destra

Successivamente si ordina ricorsivamente ciascuna delle due parti. A differenza del MergeSort, il lavoro principale avviene nella fase di **divide** (la procedura Partition), mentre la fase di **combine** è banale: una volta che le due parti sono ordinate, l'intero array è automaticamente ordinato.

7.4.1 Algoritmo

QuickSort

```
quickSort(int[] A, int p, int r){
    if(p < r){
        int q = partition(A, p, r);   // divide
        quickSort(A, p, q - 1);       // impera (parte sinistra)
        quickSort(A, q + 1, r);       // impera (parte destra)
    }
}
```

La chiamata iniziale è `quickSort(A, 1, n)` dove n è la lunghezza dell'array. Il caso base ($p \geq r$) corrisponde ad un sotto-array di zero o un elemento, che è già ordinato per definizione.

7.4.2 Procedura Partition

La procedura Partition riorganizza il sotto-array $A[p..r]$ **in loco** e restituisce un indice q tale che:

- $A[q]$ contiene il pivot nella sua posizione finale
- ogni elemento in $A[p..q-1]$ è $\leq A[q]$
- ogni elemento in $A[q+1..r]$ è $> A[q]$

Il pivot viene scelto come l'ultimo elemento $A[r]$. Si mantiene un indice i che delimita il confine della regione degli elementi $\leq x$: tutti gli elementi in $A[p..i]$ sono $\leq x$ e tutti quelli in $A[i+1..j-1]$ sono $> x$.

Partition

```
int partition(int[] A, int p, int r){
    int x = A[r];              // pivot (ultimo elemento)
    int i = p - 1;             // bordo della partizione sinistra
    int j = p;
    while(j <= r - 1){
        if(A[j] <= x){
            i := i + 1;
            // swap A[i] e A[j]
            int temp = A[i];
            A[i] := A[j];
            A[j] := temp;
        }
        j := j + 1;
    }
    // swap A[i+1] e A[r]: colloca il pivot nella posizione finale
    int temp = A[i + 1];
    A[i + 1] := A[r];
    A[r] := temp;
    return i + 1;
}
```

Esempio 7.3 – Esecuzione di Partition. Consideriamo $A = \langle 2, 8, 7, 1, 3, 5, 6, 4 \rangle$ con $p = 1$ e $r = 8$. Il pivot è $x = A[8] = 4$.

j	i	Stato di A
1	0	$\langle 2, 8, 7, 1, 3, 5, 6, 4 \rangle$ – $A[1] = 2 \leq 4$: $i := 1$, swap
2	1	$\langle 2, 8, 7, 1, 3, 5, 6, 4 \rangle$ – $A[2] = 8 > 4$: nessuno swap
3	1	$\langle 2, 8, 7, 1, 3, 5, 6, 4 \rangle$ – $A[3] = 7 > 4$: nessuno swap
4	1	$\langle 2, 1, 7, 8, 3, 5, 6, 4 \rangle$ – $A[4] = 1 \leq 4$: $i := 2$, swap
5	2	$\langle 2, 1, 3, 8, 7, 5, 6, 4 \rangle$ – $A[5] = 3 \leq 4$: $i := 3$, swap

6	3	$\langle 2,1,3,8,7,5,6,4\rangle$ – $A[6] = 5 > 4$: nessuno swap
7	3	$\langle 2,1,3,8,7,5,6,4\rangle$ – $A[7] = 6 > 4$: nessuno swap

Al termine del ciclo, $i = 3$. Si esegue lo swap di $A[4]$ con $A[8]$ (il pivot):

$$A = \langle 2,1,3,\mathbf{4},7,5,6,8\rangle$$

Partition restituisce $q = 4$. Il pivot 4 è nella sua posizione finale; tutti gli elementi a sinistra sono ≤ 4 e tutti quelli a destra sono > 4.

7.4.3 Invariante di Partition

Definizione 7.2 – Invariante del ciclo while in Partition. All'inizio di ogni iterazione del ciclo `while`, per l'indice corrente j:
1. Per ogni $k \in [p, i]$: $A[k] \leq x$ (regione degli elementi piccoli)
2. Per ogni $k \in [i+1, j-1]$: $A[k] > x$ (regione degli elementi grandi)
3. $A[r] = x$ (il pivot rimane in posizione r)

7.4.4 Correttezza di Partition

Dimostrazione. La dimostrazione procede verificando le tre proprietà dell'invariante (inizializzazione, conservazione, terminazione).

Inizializzazione ($j = p$): le regioni $A[p..i]$ e $A[i+1..j-1]$ sono vuote (poiché $i = p-1$ e $j-1 = p-1$), quindi l'invariante è banalmente soddisfatto. Il pivot $A[r] = x$ è al suo posto.

Conservazione: supponiamo che l'invariante valga all'inizio dell'iterazione j-esima. Due casi:
- Se $A[j] > x$: si incrementa solo j. L'elemento $A[j]$ entra nella regione $A[i+1..j-1]$ degli elementi $> x$, preservando l'invariante.
- Se $A[j] \leq x$: si incrementa i e si scambia $A[i]$ con $A[j]$. L'elemento che era in $A[i]$ (che era $> x$) va in posizione j, estendendo la regione $> x$; l'elemento $A[j] \leq x$ va in posizione i, estendendo la regione $\leq x$.

Terminazione ($j = r$): l'invariante garantisce che $A[p..i]$ contiene elementi $\leq x$ e $A[i+1..r-1]$ contiene elementi $> x$. Lo swap finale di $A[i+1]$ con $A[r]$ posiziona il pivot in $A[i+1]$, ottenendo la partizione corretta. ■

7.4.5 Complessità di Partition

La procedura Partition esegue un singolo scorrimento dell'array da p a $r-1$, effettuando un confronto per ogni elemento. Pertanto la sua complessità è:

$$T_{\text{partition}}(n) = \Theta(n)$$

dove $n = r - p + 1$ è il numero di elementi nel sotto-array.

7.5 Analisi di complessità di QuickSort

La complessità di QuickSort dipende interamente dal bilanciamento delle partizioni prodotte dalla scelta del pivot. La relazione di ricorrenza generale è:

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

dove q è il numero di elementi nella partizione sinistra.

7.5.1 Caso pessimo – $\Theta(n^2)$

Il caso pessimo si verifica quando le partizioni sono **massimamente sbilanciate** ad ogni chiamata ricorsiva: una partizione contiene $n-1$ elementi e l'altra 0. Questo accade, ad esempio, quando l'array è già ordinato (o ordinato in modo decrescente) e il pivot è sempre l'ultimo (o il primo) elemento.

La ricorrenza diventa:

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

Svolgendo per sostituzione:

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(n^2)$$

> **Nota – Worst case uguale a Insertion Sort.** Nel caso pessimo, QuickSort ha la stessa complessità $\Theta(n^2)$ di Insertion Sort e Selection Sort. Tuttavia, questo caso è raro in pratica e può essere evitato con la randomizzazione del pivot.

7.5.2 Caso ottimo – $\Theta(n \log n)$

Il caso ottimo si verifica quando il pivot divide sempre l'array in due parti di **uguale dimensione** (o con al più un elemento di differenza):

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

Per il Master Theorem (caso 2 con $a = 2$, $b = 2$, $f(n) = \Theta(n) = \Theta\big(n^{\log_b a}\big)$):

$$T(n) = \Theta(n \log n)$$

7.5.3 Caso medio – $\Theta(n \log n)$

> **Teorema 7.1 – Complessità nel caso medio di QuickSort.** Se tutte le permutazioni dell'input sono equiprobabili, la complessità attesa di QuickSort è $\Theta(n \log n)$.

L'intuizione fondamentale è che anche partizioni **moderatamente sbilanciate** producono un albero di ricorsione di altezza $O(\log n)$. Ad esempio, anche con uno split costante 9:1 (ogni partizione divide gli elementi in proporzione 90%-10%), la ricorrenza:

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$

si risolve comunque in $\Theta(n \log n)$, poiché l'altezza dell'albero di ricorsione è $\log_{10/9} n = O(\log n)$.

Nella pratica, è estremamente improbabile che il pivot produca sempre partizioni degeneri. Il mix di «buone» e «cattive» partizioni che si presenta nel caso medio produce comunque un comportamento $\Theta(n \log n)$.

7.6 QuickSort Randomizzato

Per eliminare la dipendenza dal caso peggiore su input particolari (ad esempio, array già ordinati), si può **randomizzare la scelta del pivot**. Invece di scegliere sempre l'ultimo elemento, si sceglie un elemento casuale uniforme nell'intervallo $[p, r]$ e lo si scambia con $A[r]$ prima di chiamare Partition.

Randomized Partition e Randomized QuickSort

```
int randomizedPartition(int[] A, int p, int r){
    int i = random(p, r);       // indice casuale uniforme in [p, r]
    // swap A[i] e A[r]
    int temp = A[i];
    A[i] := A[r];
    A[r] := temp;
    return partition(A, p, r);
}

randomizedQuickSort(int[] A, int p, int r){
    if(p < r){
        int q = randomizedPartition(A, p, r);
        randomizedQuickSort(A, p, q - 1);
        randomizedQuickSort(A, q + 1, r);
    }
}
```

> **Nota – Vantaggi della randomizzazione.** Con la randomizzazione, nessun input specifico può causare sistematicamente il caso pessimo. La complessità $\Theta(n^2)$ è ancora teoricamente possibile, ma la probabilità che si verifichi è trascurabile. Il tempo di esecuzione **atteso** è $O(n \log n)$ **per qualunque input**.

7.6.1 Analisi del numero atteso di confronti

Per analizzare formalmente il caso medio, si conta il **numero atteso di confronti** eseguiti dall'algoritmo.

Siano $z_1, z_2, ..., z_n$ gli elementi di A in ordine crescente di valore. Definiamo la variabile aleatoria indicatrice:

$$X_{ij} = \begin{cases} 1 \text{ se } z_i \text{ e } z_j \text{ vengono confrontati durante l'esecuzione} \\ 0 \text{ altrimenti} \end{cases}$$

Il numero totale di confronti è:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

> **Teorema 7.2 – Probabilità di confronto tra due elementi.**
>
> $$P(X_{ij} = 1) = \frac{2}{j - i + 1}$$
>
> **Giustificazione.** Consideriamo l'insieme $Z_{ij} = \{z_i, z_{i+1}, ..., z_j\}$ di cardinalità $j - i + 1$. Gli elementi z_i e z_j vengono confrontati se e solo se uno dei due è il **primo** elemento di Z_{ij} ad essere scelto come pivot (in una qualche chiamata ricorsiva). Se venisse scelto come pivot un qualunque elemento z_k con $i < k < j$, allora z_i e z_j finirebbero in partizioni diverse e non sarebbero mai confrontati tra loro. Poiché ogni elemento di Z_{ij} ha la stessa probabilità di essere scelto per primo come pivot, la probabilità cercata è $\frac{2}{j-i+1}$.

Applicando la linearità del valore atteso:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Con la sostituzione $k = j - i$:

$$E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = 2(n-1)H_n = O(n \log n)$$

dove $H_n = \sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$ è il numero armonico n-esimo.

7.7 Confronto degli algoritmi di ordinamento

Algoritmo	**Caso ottimo**	**Caso medio**	**Caso pessimo**	**Atteso**	**In-place**
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	–	Si
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	–	Si
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	–	No
QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	–	Si
QuickSort Rand.	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$	Si

Tabella 10: Confronto delle complessità in tempo degli algoritmi di ordinamento basati su confronti. La colonna «Atteso» indica la complessità attesa del QuickSort Randomizzato: il caso pessimo deterministico resta $\Theta(n^2)$ ma si verifica con probabilità trascurabile.

7.8 Heap e HeapSort

7.8.1 Definizione di Heap

Definizione 7.3 – Albero binario quasi completo. Un albero binario è **quasi completo** (o *nearly complete*) se tutti i livelli sono completamente riempiti, eccetto eventualmente l'ultimo, che deve essere riempito da sinistra verso destra senza interruzioni.

Definizione 7.4 – Heap binario. Un **heap binario** è un albero binario quasi completo che soddisfa la **proprietà di heap**:

- **Max-Heap**: per ogni nodo i diverso dalla radice vale $A[\text{Parent}(i)] \geq A[i]$, ossia ogni nodo ha chiave non superiore a quella del padre.
- **Min-Heap**: per ogni nodo i diverso dalla radice vale $A[\text{Parent}(i)] \leq A[i]$, ossia ogni nodo ha chiave non inferiore a quella del padre.

Nel seguito tratteremo esclusivamente il caso del max-heap; le considerazioni per il min-heap sono simmetriche.

Da queste definizioni seguono immediatamente alcune proprietà fondamentali.

Teorema 7.3 – Proprietà strutturali di un max-heap. Sia $A[1..n]$ un max-heap con n elementi. Allora:

1. L'elemento massimo si trova nella radice: $A[1] \geq A[i]$ per ogni $i \in \{1, ..., n\}$.
2. L'altezza dell'heap è $h = \lfloor \log_2 n \rfloor$, dunque $h = \Theta(\log n)$.
3. Il numero di nodi a un'altezza h' (contata dalle foglie) è al massimo $\left\lceil n/2^{h'+1} \right\rceil$.

4. Le foglie occupano le posizioni $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, ..., n$.

Dimostrazione. (1) Segue direttamente dalla proprietà di max-heap applicata ripetutamente lungo il cammino dalla radice a qualsiasi nodo.

(2) In un albero binario quasi completo con n nodi, il livello ℓ (con la radice al livello 0) contiene al massimo 2^ℓ nodi. Poiché tutti i livelli fino a $h-1$ sono pieni, si ha $n \geq 2^h$, da cui $h \leq \log_2 n$. Inoltre $n < 2^{h+1}$, da cui $h > \log_2 n - 1$. Pertanto $h = \lfloor \log_2 n \rfloor$.

(3) Sia $h' \in \{0, 1, ..., h\}$ un'altezza misurata dalle foglie. I nodi a altezza h' si trovano al livello $h - h'$. Poiché l'heap ha al massimo n nodi e il numero di nodi fino al livello $h - h' - 1$ vale $2^{h-h'} - 1$, i nodi al livello $h - h'$ sono al massimo $\left\lceil n/2^{h'+1} \right\rceil$.

(4) Un nodo in posizione i ha figlio sinistro in posizione $2i$. Se $2i > n$, allora il nodo non ha figli ed è dunque una foglia. Questo accade per $i > \lfloor n/2 \rfloor$. ■

7.8.2 Rappresentazione implicita come array

Un heap puo essere memorizzato in modo compatto in un array $A[1..n]$, senza bisogno di puntatori espliciti. La radice si trova in $A[1]$ e, per un nodo in posizione i:

$$\text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor, \quad \text{Left}(i) = 2i, \quad \text{Right}(i) = 2i + 1$$

Navigazione nell'heap

```
int parent(int i){ return i / 2; }
int left(int i)  { return 2 * i; }
int right(int i) { return 2 * i + 1; }
```

Nota – Efficienza della rappresentazione. Questa rappresentazione implicita presenta diversi vantaggi:

- **Nessun puntatore**: la relazione padre-figlio e codificata dall'aritmetica sugli indici, con notevole risparmio di spazio.
- **Navigazione in** $O(1)$: le operazioni `parent`, `left` e `right` sono semplici divisioni e moltiplicazioni intere.
- **Localita in memoria**: gli elementi sono memorizzati in posizioni contigue, il che favorisce le prestazioni della cache.

Esempio 7.4 – Rappresentazione array di un max-heap. Consideriamo l'array $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ con $n = 10$.

Esso corrisponde al seguente max-heap:

- Radice: $A[1] = 16$
- Figli di 16: $A[2] = 14$ (sinistro), $A[3] = 10$ (destro)
- Figli di 14: $A[4] = 8$ (sinistro), $A[5] = 7$ (destro)
- Figli di 10: $A[6] = 9$ (sinistro), $A[7] = 3$ (destro)
- Figli di 8: $A[8] = 2$ (sinistro), $A[9] = 4$ (destro)
- Figlio di 7: $A[10] = 1$ (sinistro; il destro non esiste)

Si puo verificare che la proprieta di max-heap vale per ogni nodo: $A[\text{Parent}(i)] \geq A[i]$ per $i = 2, ..., 10$.

7.8.3 Max-Heapify

La procedura `Max-Heapify` e il mattone fondamentale su cui si costruiscono tutte le operazioni sugli heap. Essa ripristina la proprieta di max-heap nel sottoalbero radicato in posizione i, sotto l'ipotesi che i sottoalberi sinistro e destro di i siano gia max-heap validi. L'unica possibile violazione si trova dunque nel nodo i stesso.

L'idea e semplice: si confronta $A[i]$ con i suoi due figli; se uno dei figli e maggiore, lo si scambia con $A[i]$ e si prosegue ricorsivamente verso il basso (*sift-down*).

Max-Heapify

```
maxHeapify(int[] A, int i, int heapsize){
    int l = left(i);
    int r = right(i);
    int largest = i;

    // Trova il massimo tra A[i], A[l], A[r]
    if((l <= heapsize) && (A[l] > A[i])){
        largest := l;
    }

    if((r <= heapsize) && (A[r] > A[largest])){
        largest := r;
    }

    // Se il massimo non e il nodo corrente, scambia e ricorri
    if(largest != i){
        int temp = A[i];
        A[i] := A[largest];
        A[largest] := temp;
        maxHeapify(A, largest, heapsize);
    }
}
```

7.8.3.1 Correttezza di Max-Heapify

> **Teorema 7.4 – Correttezza di Max-Heapify.** Se i sottoalberi radicati in Left(i) e Right(i) sono max-heap, allora dopo la chiamata `Max-Heapify(A, i, heapsize)` il sottoalbero radicato in i e un max-heap.

Dimostrazione. Procediamo per induzione sull'altezza h del nodo i nell'heap.

Caso base ($h = 0$): il nodo i e una foglia, non ha figli, e dunque e banalmente radice di un max-heap. La procedura non esegue alcuno scambio.

Passo induttivo ($h > 0$): supponiamo che `Max-Heapify` funzioni correttamente per ogni nodo di altezza minore di h. La procedura calcola l'indice `largest` del massimo tra $A[i]$, $A[l]$ e $A[r]$.

- Se largest $= i$, allora $A[i] \geq A[l]$ e $A[i] \geq A[r]$. Poiche i sottoalberi di l e r sono max-heap per ipotesi, il sottoalbero radicato in i e gia un max-heap.
- Se largest $\neq i$, si scambia $A[i]$ con A[largest]. Dopo lo scambio, il nodo i soddisfa la proprieta di max-heap rispetto ai suoi figli. Il sottoalbero radicato in `largest` potrebbe pero violare la proprieta, ma ha altezza al piu $h - 1$. Per ipotesi induttiva, la chiamata ricorsiva `Max-Heapify(A, largest, heapsize)` ripristina la proprieta.

■

7.8.3.2 Complessita di Max-Heapify

Teorema 7.5 – Complessita di Max-Heapify. La procedura `Max-Heapify` su un nodo di altezza h ha complessita $O(h)$. Per un heap di n elementi, nel caso pessimo si ha:

$$T(n) = O(\log n)$$

Dimostrazione. A ogni chiamata ricorsiva, la procedura scende di un livello nell'heap. Il numero massimo di livelli che puo attraversare e l'altezza h del nodo di partenza. Poiche ogni livello richiede lavoro $O(1)$ (un confronto e un eventuale scambio), il costo totale e $O(h)$.

L'altezza massima di un nodo nell'heap e $\lfloor \log_2 n \rfloor$, da cui $T(n) = O(\log n)$. ■

Esempio 7.5 – Max-Heapify passo-passo. Consideriamo l'array $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$ con heapsize $= 10$ e invochiamo `Max-Heapify(A, 2, 10)`.

Il nodo in posizione 2 ha valore 4. I sottoalberi sinistro (radice $A[4] = 14$) e destro (radice $A[5] = 7$) sono gia max-heap, ma $A[2] = 4 < A[4] = 14$, quindi la proprieta di max-heap e violata in posizione 2.

Passo 1: confronto in posizione 2
- Nodo corrente: $A[2] = 4$
- Figlio sinistro: $A[4] = 14$
- Figlio destro: $A[5] = 7$
- largest $= 4$ (poiche $14 > 4$ e $14 > 7$)
- Scambio $A[2] \leftrightarrow A[4]$

Array dopo il passo 1: $[16, \mathbf{14}, 10, \mathbf{4}, 7, 9, 3, 2, 8, 1]$

Passo 2: chiamata ricorsiva in posizione 4
- Nodo corrente: $A[4] = 4$
- Figlio sinistro: $A[8] = 2$
- Figlio destro: $A[9] = 8$
- largest $= 9$ (poiche $8 > 4$ e $8 > 2$)
- Scambio $A[4] \leftrightarrow A[9]$

Array dopo il passo 2: $[16, 14, 10, \mathbf{8}, 7, 9, 3, 2, \mathbf{4}, 1]$

Passo 3: chiamata ricorsiva in posizione 9
- Nodo corrente: $A[9] = 4$
- Left$(9) = 18 >$ heapsize: nessun figlio
- Il nodo e una foglia: la procedura termina

Risultato finale: $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$

Il valore 4 e «sceso» dalla posizione 2 alla posizione 9, attraversando 2 scambi (altezza percorsa = 2).

7.8.4 Build-Max-Heap

Data un array arbitrario $A[1..n]$, la procedura `Build-Max-Heap` lo trasforma in un max-heap. L'idea e applicare `Max-Heapify` a tutti i nodi interni, procedendo dal basso verso l'alto (dalle foglie verso la radice). Poiche le foglie sono gia heap banali (sottalberi di un solo nodo), si parte dalla posizione $\lfloor n/2 \rfloor$ e si scende fino a 1.

Build-Max-Heap

```
buildMaxHeap(int[] A, int n){
    int heapsize = n;
    int i = n / 2;
    while(i >= 1){
        maxHeapify(A, i, heapsize);
        i := i - 1;
    }
}
```

Nota. Si parte da $i = \lfloor n/2 \rfloor$ e non da $i = n$ perche i nodi nelle posizioni $\lfloor n/2 \rfloor + 1, ..., n$ sono foglie (non hanno figli) e sono pertanto max-heap banali di un singolo elemento. Iniziare da essi sarebbe corretto ma inutile.

7.8.4.1 Correttezza di Build-Max-Heap

Definizione 7.5 – Invariante di ciclo per Build-Max-Heap. All'inizio di ogni iterazione del ciclo `while`, ogni nodo $j \in \{i+1, i+2, ..., n\}$ e radice di un max-heap.

Dimostrazione. **Inizializzazione**: prima della prima iterazione si ha $i = \lfloor n/2 \rfloor$. I nodi $\lfloor n/2 \rfloor + 1, ..., n$ sono foglie, dunque radici di max-heap banali. L'invariante vale.

Mantenimento: all'iterazione corrente, l'invariante garantisce che i figli del nodo i (che si trovano nelle posizioni $2i$ e $2i+1$, entrambe $> i$) sono radici di max-heap. Questo e esattamente il prerequisito di `Max-Heapify(A, i, heapsize)`, che rende il sottoalbero radicato in i un max-heap. Dopo il decremento $i := i - 1$, l'invariante si estende al nodo $i + 1$ appena trattato.

Terminazione: il ciclo termina quando $i = 0$. A quel punto l'invariante afferma che ogni nodo $j \in \{1, 2, ..., n\}$ e radice di un max-heap. In particolare, il nodo 1 (la radice dell'intero albero) e radice di un max-heap, il che significa che l'intero array e un max-heap. ■

7.8.4.2 Complessita di Build-Max-Heap

7.8.4.2.1 Analisi ingenua

Si effettuano $O(n)$ chiamate a `Max-Heapify`, ciascuna di costo $O(\log n)$. Questo porta a un limite superiore di $O(n \log n)$, che e tuttavia non stretto.

7.8.4.2.2 Analisi stretta

Teorema 7.6 – Complessita di Build-Max-Heap. La procedura `Build-Max-Heap` ha complessita $\Theta(n)$.

Dimostrazione. L'altezza dell'heap e $h = \lfloor \log_2 n \rfloor$. Il costo di `Max-Heapify` su un nodo a altezza h' (misurata dalle foglie) e $O(h')$.

Per la proprieta (3) degli heap, a altezza h' vi sono al massimo $\left\lceil n/2^{h'+1} \right\rceil$ nodi. Sommando su tutte le altezze:

$$T(n) = \sum_{h'=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h'+1}} \right\rceil \cdot O(h') = O\left(n \sum_{h'=0}^{\lfloor \log_2 n \rfloor} \frac{h'}{2^{h'}} \right)$$

La serie $\sum_{h'=0}^{\infty} \frac{h'}{2^{h'}}$ converge e vale esattamente 2, essendo derivata della serie geometrica. Pertanto:

$$T(n) = O(n \cdot 2) = O(n)$$

Il limite inferiore $\Omega(n)$ segue banalmente dal fatto che ogni elemento dell'array deve essere esaminato almeno una volta. Dunque $T(n) = \Theta(n)$. ■

Nota – Intuizione della complessita lineare. Il risultato $\Theta(n)$ puo sembrare sorprendente, poiche si eseguono $O(n)$ chiamate a `Max-Heapify`, ciascuna potenzialmente $O(\log n)$. Il punto chiave e che la *maggior parte* dei nodi ha altezza piccola: circa $n/2$ nodi sono foglie (altezza 0), circa $n/4$ hanno altezza 1, e cosi via. Solo un nodo (la radice) ha altezza $\lfloor \log n \rfloor$. Il lavoro effettivo e quindi dominato dai nodi bassi.

Esempio 7.6 – Build-Max-Heap passo-passo. Costruiamo un max-heap dall'array $A = [7, 4, 3, 8, 9, 6]$ con $n = 6$.

Calcoliamo $\lfloor n/2 \rfloor = 3$: partiamo da $i = 3$ e procediamo fino a $i = 1$.

Stato iniziale:

Array: $[7, 4, 3, 8, 9, 6]$ (indici $1, ..., 6$)

Iterazione $i = 3$: Max-Heapify(A, 3, 6)
- Nodo: $A[3] = 3$
- Figlio sinistro: $A[6] = 6$, figlio destro: non esiste
- largest $= 6$ (poiche $6 > 3$), scambio $A[3] \leftrightarrow A[6]$

Array: $[7, 4, \mathbf{6}, 8, 9, \mathbf{3}]$

Iterazione $i = 2$: Max-Heapify(A, 2, 6)
- Nodo: $A[2] = 4$
- Figlio sinistro: $A[4] = 8$, figlio destro: $A[5] = 9$
- largest $= 5$ (poiche $9 > 4$ e $9 > 8$), scambio $A[2] \leftrightarrow A[5]$
- Chiamata ricorsiva su posizione 5: e una foglia, termina

Array: $[7, \mathbf{9}, 6, 8, \mathbf{4}, 3]$

Iterazione $i = 1$: Max-Heapify(A, 1, 6)
- Nodo: $A[1] = 7$
- Figlio sinistro: $A[2] = 9$, figlio destro: $A[3] = 6$
- largest $= 2$ (poiche $9 > 7$ e $9 > 6$), scambio $A[1] \leftrightarrow A[2]$

Array intermedio: $[\mathbf{9}, \mathbf{7}, 6, 8, 4, 3]$

- Chiamata ricorsiva su posizione 2:
  - Nodo: $A[2] = 7$
  - Figlio sinistro: $A[4] = 8$, figlio destro: $A[5] = 4$
  - largest $= 4$ (poiche $8 > 7$), scambio $A[2] \leftrightarrow A[4]$

Array: $[9, \mathbf{8}, 6, \mathbf{7}, 4, 3]$

- Chiamata ricorsiva su posizione 4: figli $A[8]$ e $A[9]$ non esistono, termina

Risultato finale: $A = [9, 8, 6, 7, 4, 3]$ – un max-heap valido.

Iterazione	i	**Operazione**	**Array risultante**
Iniziale	-	-	$[7, 4, 3, 8, 9, 6]$
1	3	scambio $3 \leftrightarrow 6$	$[7, 4, 6, 8, 9, 3]$

2	2	scambio $4 \leftrightarrow 9$	$[7, 9, 6, 8, 4, 3]$
3	1	scambio $7 \leftrightarrow 9$, poi $7 \leftrightarrow 8$	$[9, 8, 6, 7, 4, 3]$

7.8.5 HeapSort

L'algoritmo HeapSort sfrutta la struttura del max-heap per ordinare un array in loco.

7.8.5.1 Idea dell'algoritmo

1. Si costruisce un max-heap dall'array con `Build-Max-Heap`. A questo punto il massimo e in $A[1]$.
2. Si scambia $A[1]$ (il massimo) con l'ultimo elemento dell'heap $A[\text{heapsize}]$.
3. Si riduce `heapsize` di 1: l'elemento appena spostato in fondo si trova nella sua posizione finale.
4. Si invoca `Max-Heapify(A, 1, heapsize)` per ripristinare la proprieta di max-heap sulla parte rimanente.
5. Si ripetono i passi 2–4 finche `heapsize` vale 1.

HeapSort

```
heapsort(int[] A, int n){
    buildMaxHeap(A, n);
    int heapsize = n;
    int i = n;
    while(i >= 2){
        // swap(A[1], A[i]) - metti il massimo in fondo
        int temp = A[1];
        A[1] := A[i];
        A[i] := temp;

        heapsize := heapsize - 1;
        maxHeapify(A, 1, heapsize);
        i := i - 1;
    }
}
```

7.8.5.2 Correttezza di HeapSort

Definizione 7.6 – Invariante di ciclo per HeapSort. All'inizio di ogni iterazione del ciclo `while` con indice i:
1. $A[1..i]$ e un max-heap contenente gli i elementi piu piccoli di $A[1..n]$.
2. $A[i+1..n]$ contiene gli $n-i$ elementi piu grandi di $A[1..n]$, in ordine crescente.

Dimostrazione. **Inizializzazione**: prima della prima iterazione, $i = n$. L'array $A[1..n]$ e un max-heap (appena costruito da `Build-Max-Heap`) e $A[n+1..n]$ e vuoto. L'invariante vale banalmente.

Mantenimento: all'inizio dell'iterazione con indice i, $A[1]$ contiene il massimo di $A[1..i]$ (proprieta del max-heap). Scambiandolo con $A[i]$, l'elemento piu grande finisce in posizione i. Ora $A[i..n]$ contiene gli $n-i+1$ elementi piu grandi in ordine crescente. Dopo il decremento di `heapsize` e la chiamata a `Max-Heapify(A, 1, heapsize)`, il sotto-array $A[1..i-1]$ e nuovamente un max-heap. L'invariante vale con $i-1$.

Terminazione: quando $i = 1$, l'invariante afferma che $A[2..n]$ contiene gli $n-1$ elementi piu grandi in ordine crescente, e $A[1]$ e il minimo. L'array e ordinato. ■

7.8.5.3 Complessita di HeapSort

Teorema 7.7 – Complessita di HeapSort. HeapSort ha complessita $\Theta(n \log n)$ nel caso pessimo, ottimo e medio.

Dimostrazione.

$$T(n) = \underbrace{\Theta(n)}_{\text{Build-Max-Heap}} + \underbrace{(n-1) \cdot O(\log n)}_{\text{ciclo principale}} = O(n \log n)$$

La chiamata a `Build-Max-Heap` costa $\Theta(n)$. Il ciclo esegue $n-1$ iterazioni, ciascuna con una chiamata a `Max-Heapify` sulla radice di un heap di dimensione decrescente, per un costo di $O(\log n)$ per iterazione. Questo stabilisce il limite superiore $O(n \log n)$.

Per il limite inferiore, si osserva che qualsiasi algoritmo di ordinamento basato su confronti richiede $\Omega(n \log n)$ confronti nel caso pessimo (limite inferiore informazionale). Pertanto $T(n) = \Theta(n \log n)$.■

Nota – Caratteristiche di HeapSort.

- **Complessita**: $\Theta(n \log n)$ nel caso pessimo, ottimo e medio. A differenza di QuickSort, non ha un caso pessimo $O(n^2)$.
- **In-place**: utilizza solo una quantita costante di memoria aggiuntiva (nessun array ausiliario).
- **Non stabile**: lo scambio tra radice e ultimo elemento puo alterare l'ordine relativo di elementi con chiave uguale.
- Nella pratica, QuickSort e spesso piu veloce di HeapSort a causa di una migliore localita di cache, nonostante il caso pessimo peggiore.

Esempio 7.7 – HeapSort passo-passo. Ordiniamo l'array $A = [7, 4, 3, 8, 9, 6]$ usando HeapSort.

Fase 1: Build-Max-Heap

Come mostrato nell'esempio precedente, dopo `Build-Max-Heap` si ottiene:

$$A = [9, 8, 6, 7, 4, 3] \text{ con heapsize} = 6$$

Fase 2: estrazione iterativa del massimo

Iter.	heapsize	Scambio	Dopo scambio	Dopo Max-Heapify
1	6	$A[1] \leftrightarrow A[6]$	$[3, 8, 6, 7, 4 \mid 9]$	$[8, 7, 6, 3, 4 \mid 9]$
2	5	$A[1] \leftrightarrow A[5]$	$[4, 7, 6, 3 \mid 8, 9]$	$[7, 4, 6, 3 \mid 8, 9]$
3	4	$A[1] \leftrightarrow A[4]$	$[3, 4, 6 \mid 7, 8, 9]$	$[6, 4, 3 \mid 7, 8, 9]$
4	3	$A[1] \leftrightarrow A[3]$	$[3, 4 \mid 6, 7, 8, 9]$	$[4, 3 \mid 6, 7, 8, 9]$
5	2	$A[1] \leftrightarrow A[2]$	$[3 \mid 4, 6, 7, 8, 9]$	$[3 \mid 4, 6, 7, 8, 9]$

La barra verticale «|» separa la parte heap (sinistra) dalla parte gia ordinata (destra).

Dettaglio iterazione 1 ($i = 6$):

- Scambio $A[1] = 9$ con $A[6] = 3$: array $= [3, 8, 6, 7, 4, 9]$
- Riduco `heapsize` a 5 (il 9 e nella sua posizione finale)
- `Max-Heapify(A, 1, 5)`:
  - $A[1] = 3$, figli: $A[2] = 8$, $A[3] = 6$. Massimo: 8, scambio $A[1] \leftrightarrow A[2]$
  - Risultato intermedio: $[8, 3, 6, 7, 4, 9]$
  - Ricorsione su posizione 2: $A[2] = 3$, figli: $A[4] = 7$, $A[5] = 4$. Massimo: 7, scambio $A[2] \leftrightarrow A[4]$

- Risultato: $[8, 7, 6, 3, 4, 9]$

Dettaglio iterazione 2 ($i = 5$):
- Scambio $A[1] = 8$ con $A[5] = 4$: array $= [4, 7, 6, 3, 8, 9]$
- Riduco `heapsize` a 4
- `Max-Heapify(A, 1, 4)`:
  - $A[1] = 4$, figli: $A[2] = 7$, $A[3] = 6$. Massimo: 7, scambio $A[1] \leftrightarrow A[2]$
  - Risultato: $[7, 4, 6, 3, 8, 9]$
  - Ricorsione su posizione 2: $A[2] = 4$, figli: $A[4] = 3$, nessun figlio destro. $4 > 3$: nessuno scambio

Dettaglio iterazione 3 ($i = 4$):
- Scambio $A[1] = 7$ con $A[4] = 3$: array $= [3, 4, 6, 7, 8, 9]$
- Riduco `heapsize` a 3
- `Max-Heapify(A, 1, 3)`:
  - $A[1] = 3$, figli: $A[2] = 4$, $A[3] = 6$. Massimo: 6, scambio $A[1] \leftrightarrow A[3]$
  - Risultato: $[6, 4, 3, 7, 8, 9]$

Dettaglio iterazione 4 ($i = 3$):
- Scambio $A[1] = 6$ con $A[3] = 3$: array $= [3, 4, 6, 7, 8, 9]$
- Riduco `heapsize` a 2
- `Max-Heapify(A, 1, 2)`:
  - $A[1] = 3$, figlio sinistro: $A[2] = 4$. Scambio $A[1] \leftrightarrow A[2]$
  - Risultato: $[4, 3, 6, 7, 8, 9]$

Dettaglio iterazione 5 ($i = 2$):
- Scambio $A[1] = 4$ con $A[2] = 3$: array $= [3, 4, 6, 7, 8, 9]$
- Riduco `heapsize` a 1
- `Max-Heapify(A, 1, 1)`: un solo elemento, nulla da fare

Risultato finale: $A = [3, 4, 6, 7, 8, 9]$ – l'array e ordinato in ordine crescente.

7.8.6 Code di priorita

Una coda di priorita e una struttura dati astratta che mantiene una collezione dinamica di elementi, ciascuno dotato di una **chiave** (o *priorita*), e supporta operazioni efficienti di inserimento ed estrazione dell'elemento con chiave massima (o minima). Le code di priorita sono alla base di numerosi algoritmi fondamentali.

Definizione 7.7 – Coda di priorita (max). Una **coda di priorita max** e una struttura dati che mantiene un insieme dinamico S di elementi, ciascuno con una chiave associata, e supporta le seguenti operazioni:
- `Maximum(S)`: restituisce l'elemento di S con chiave massima, senza modificare S.
- `Extract-Max(S)`: rimuove e restituisce l'elemento di S con chiave massima.
- `Increase-Key(S, x, k)`: aumenta la chiave dell'elemento x al nuovo valore $k \geq A[x]$.
- `Insert(S, x)`: inserisce un nuovo elemento x in S.

7.8.6.1 Applicazioni

Le code di priorita trovano impiego in numerosi contesti:
- **Scheduling di processi**: il sistema operativo seleziona il processo con priorita piu alta da eseguire.
- **Simulazione di eventi discreti**: gli eventi vengono estratti in ordine cronologico (priorita = tempo dell'evento).

- **Algoritmo di Dijkstra**: coda di priorita min per estrarre il vertice con distanza stimata minima.
- **Algoritmo di Prim**: per la costruzione dell'albero di copertura minimo (MST).

7.8.6.2 Implementazione con Max-Heap

Un max-heap fornisce un'implementazione naturale ed efficiente di una coda di priorita max.

7.8.6.2.1 Maximum

L'operazione piu semplice: il massimo e sempre la radice dell'heap.

Heap-Maximum

```
int heapMaximum(int[] A){
    return A[1];
}
```

Complessita: $O(1)$.

7.8.6.2.2 Extract-Max

Si salva il valore della radice, si sostituisce la radice con l'ultimo elemento dell'heap, si riduce la dimensione e si invoca `Max-Heapify` per ripristinare la proprieta.

Heap-Extract-Max

```
int heapExtractMax(int[] A, int heapsize){
    if(heapsize < 1){
        // errore: heap vuoto (underflow)
        return -1;
    }
    int max = A[1];
    A[1] := A[heapsize];
    heapsize := heapsize - 1;
    maxHeapify(A, 1, heapsize);
    return max;
}
```

Complessita: $O(\log n)$, dominata dalla chiamata a `Max-Heapify`.

7.8.6.2.3 Increase-Key

Per aumentare la chiave di un elemento, si aggiorna il valore e si fa «risalire» l'elemento verso la radice (*sift-up*), scambiandolo con il padre finche la proprieta di max-heap non e ripristinata.

Heap-Increase-Key

```
heapIncreaseKey(int[] A, int i, int key){
    if(key < A[i]){
        // errore: la nuova chiave e minore della precedente
        return;
    }
    A[i] := key;
    // Risali verso la radice finche necessario
    while((i > 1) && (A[parent(i)] < A[i])){
        int temp = A[i];
        A[i] := A[parent(i)];
        A[parent(i)] := temp;
        i := parent(i);
```

```
    }
}
```

Complessita: $O(\log n)$. Nel caso pessimo l'elemento risale dalla foglia fino alla radice, percorrendo $O(\log n)$ livelli.

7.8.6.2.4 Insert

Per inserire un nuovo elemento, si estende l'heap di una posizione in fondo, si inserisce un valore «sentinella» $-\infty$ e si invoca `Increase-Key` per portare la chiave al valore desiderato.

Heap-Insert

```
heapInsert(int[] A, int key, int heapsize){
    heapsize := heapsize + 1;
    A[heapsize] := -∞;
    heapIncreaseKey(A, heapsize, key);
}
```

Complessita: $O(\log n)$, dominata dalla chiamata a `Increase-Key`.

Esempio 7.8 – Operazioni su una coda di priorita. Partiamo dal max-heap $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ con heapsize $= 10$.

Operazione 1: Maximum
- Restituisce $A[1] = 16$. L'heap non cambia.

Operazione 2: Extract-Max
- Si salva max $= A[1] = 16$
- Si pone $A[1] \coloneqq A[10] = 1$, si riduce `heapsize` a 9
- Array: $[1, 14, 10, 8, 7, 9, 3, 2, 4]$
- `Max-Heapify(A, 1, 9)`:
  - $A[1] = 1$, figli: 14, 10. Scambio con 14 (posizione 2)
  - $A[2] = 1$, figli: 8, 7. Scambio con 8 (posizione 4)
  - $A[4] = 1$, figli: 2, 4. Scambio con 4 (posizione 9)
- Risultato: $A = [14, 8, 10, 4, 7, 9, 3, 2, 1]$, restituisce 16

Operazione 3: Increase-Key(A, 9, 15) (aumento la chiave in posizione 9 a 15)
- Si pone $A[9] \coloneqq 15$
- Array: $[14, 8, 10, 4, 7, 9, 3, 2, 15]$
- Risalita: $A[9] = 15 > A[4] = 4$, scambio. Array: $[14, 8, 10, 15, 7, 9, 3, 2, 4]$
- Risalita: $A[4] = 15 > A[2] = 8$, scambio. Array: $[14, 15, 10, 8, 7, 9, 3, 2, 4]$
- Risalita: $A[2] = 15 > A[1] = 14$, scambio. Array: $[15, 14, 10, 8, 7, 9, 3, 2, 4]$
- $i = 1$: radice raggiunta, terminazione

Operazione 4: Insert(A, 12) con heapsize $= 9$
- Si pone heapsize $\coloneqq 10$, $A[10] \coloneqq -\infty$
- `Increase-Key(A, 10, 12)`: $A[10] \coloneqq 12$
- Risalita: $A[10] = 12 > A[5] = 7$, scambio. Array con $A[5] = 12$, $A[10] = 7$
- Risalita: $A[5] = 12 < A[2] = 14$: nessuno scambio, terminazione
- Risultato: $A = [15, 14, 10, 8, 12, 9, 3, 2, 4, 7]$

7.8.6.3 Riepilogo complessita

Operazione	Max-Heap	Array non ordinato
`Maximum`	$O(1)$	$O(n)$
`Extract-Max`	$O(\log n)$	$O(n)$
`Insert`	$O(\log n)$	$O(1)$
`Increase-Key`	$O(\log n)$	$O(1)$

Tabella 11: Confronto delle complessita per le operazioni sulle code di priorita

> **Nota – Compromesso dell'implementazione con heap.** L'implementazione con max-heap offre un buon compromesso: tutte e quattro le operazioni hanno costo al piu logaritmico. Con un array non ordinato, `Insert` e `Increase-Key` sono $O(1)$, ma `Maximum` ed `Extract-Max` richiedono una scansione lineare $O(n)$. Simmetricamente, con un array ordinato `Maximum` ed `Extract-Max` sono $O(1)$, ma l'inserimento richiede $O(n)$ per mantenere l'ordinamento.

7.9 Limite inferiore per l'ordinamento basato su confronti

> **Nota.** La teoria generale dei limiti inferiori alla difficoltà dei problemi (criterio della dimensione dell'input, criterio dell'albero di decisione, criterio degli eventi contabili) è trattata nel capitolo sulla complessità computazionale. In questa sezione applichiamo quei risultati al caso specifico dell'ordinamento.

Tutti gli algoritmi di ordinamento studiati finora (Insertion Sort, Selection Sort, MergeSort, QuickSort, HeapSort) si basano esclusivamente su **confronti** tra coppie di elementi per determinare l'ordinamento. Una domanda naturale è: esiste un limite inferiore alla complessità di qualsiasi algoritmo di questo tipo?

> **Teorema 7.8 – Limite inferiore per l'ordinamento basato su confronti.** Qualsiasi algoritmo di ordinamento basato su confronti richiede $\Omega(n \log n)$ confronti nel caso pessimo per ordinare n elementi.

Per dimostrare questo risultato, si introduce il modello dell'**albero di decisione**.

7.9.1 Albero di decisione

> **Definizione 7.8 – Albero di decisione.** Un **albero di decisione** è un albero binario completo che modella tutti i possibili confronti effettuati da un algoritmo di ordinamento su un input di dimensione n:
> - Ogni **nodo interno** rappresenta un confronto del tipo $a_i \leq a_j$, dove i, j sono indici degli elementi.
> - Il **sottoalbero sinistro** di un nodo corrisponde all'esecuzione nel caso in cui il confronto dia esito positivo ($a_i \leq a_j$).
> - Il **sottoalbero destro** corrisponde al caso in cui il confronto dia esito negativo ($a_i > a_j$).
> - Ogni **foglia** rappresenta una permutazione dell'output, cioè un possibile ordinamento finale degli elementi.
>
> L'altezza dell'albero corrisponde al numero massimo di confronti effettuati dall'algoritmo nel caso pessimo.

7.9.2 Dimostrazione del limite inferiore

Dimostrazione. Sia h l'altezza dell'albero di decisione associato a un algoritmo che ordina n elementi.

Osservazione 1. L'algoritmo deve essere in grado di produrre ogni possibile permutazione come output. Poiché un array di n elementi distinti ammette $n!$ permutazioni, l'albero deve avere almeno $n!$ foglie.

Osservazione 2. Un albero binario di altezza h ha al più 2^h foglie.

Combinando le due osservazioni:

$$2^h \geq n!$$

$$h \geq \log_2(n!)$$

Applicando l'**approssimazione di Stirling** ($n! \approx \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$), si ottiene:

$$\log_2(n!) = \sum_{i=1}^{n} \log_2(i) \geq \sum_{i=\lceil n/2 \rceil}^{n} \log_2(i) \geq \frac{n}{2} \cdot \log_2\left(\frac{n}{2}\right) = \Omega(n \log n)$$

Pertanto:

$$h = \Omega(n \log n)$$

Qualsiasi algoritmo basato su confronti deve eseguire almeno $\Omega(n \log n)$ confronti nel caso pessimo.■

> **Nota – Conseguenze del limite inferiore.**
> - MergeSort e HeapSort, avendo complessità $\Theta(n \log n)$ nel caso pessimo, sono **asintoticamente ottimi** tra gli algoritmi basati su confronti.
> - QuickSort ha caso pessimo $O(n^2)$, ma caso medio $O(n \log n)$.
> - Per superare la barriera $\Omega(n \log n)$ è necessario **non basarsi esclusivamente su confronti**, sfruttando informazioni aggiuntive sulla struttura dei dati (ad esempio, il fatto che siano interi in un intervallo limitato).

7.10 Ordinamento in tempo lineare

Gli algoritmi presentati di seguito raggiungono complessità lineare o quasi-lineare sfruttando ipotesi aggiuntive sull'input. Non si basano esclusivamente su confronti tra coppie di elementi e quindi non sono soggetti al limite inferiore $\Omega(n \log n)$.

7.10.1 Counting Sort

Il **Counting Sort** è un algoritmo di ordinamento che opera su interi nell'intervallo $[0, k]$. L'idea fondamentale è **contare** il numero di occorrenze di ciascun valore e utilizzare queste informazioni per determinare direttamente la posizione finale di ogni elemento nell'array ordinato.

7.10.1.1 Idea

L'algoritmo si articola in quattro fasi:

1. **Inizializzazione**: si crea un array ausiliario $C[0...k]$ inizializzato a zero.
2. **Conteggio**: si scorre l'array di input A e per ogni elemento $A[j]$ si incrementa $C[A[j]]$. Al termine, $C[i]$ contiene il numero di elementi uguali a i.
3. **Somme prefisse**: si calcolano le somme prefisse su C, cioè $C[i] \coloneqq C[i] + C[i-1]$. Dopo questa fase, $C[i]$ contiene il numero di elementi $\leq i$, e quindi indica la posizione finale dell'ultimo elemento con valore i nell'array ordinato.

4. **Costruzione dell'output**: si scorre A da destra a sinistra. Per ogni elemento $A[j]$, si piazza $A[j]$ nella posizione $C[A[j]]$ dell'array di output B, e si decrementa $C[A[j]]$.

7.10.1.2 Algoritmo

Counting Sort

```
countingSort(int[] A, int[] B, int n, int k){
    // Fase 1: Inizializzazione di C
    int[] C = new int[k + 1];
    int i = 0;
    while(i <= k){
        C[i] := 0;
        i := i + 1;
    }

    // Fase 2: Conteggio delle occorrenze
    int j = 1;
    while(j <= n){
        C[A[j]] := C[A[j]] + 1;
        j := j + 1;
    }

    // Fase 3: Somme prefisse
    i := 1;
    while(i <= k){
        C[i] := C[i] + C[i - 1];
        i := i + 1;
    }

    // Fase 4: Costruzione dell'output (da destra a sinistra)
    j := n;
    while(j >= 1){
        B[C[A[j]]] := A[j];
        C[A[j]] := C[A[j]] - 1;
        j := j - 1;
    }
}
```

7.10.1.3 Esempio passo-passo

Esempio 7.9 – Esecuzione di Counting Sort. Input: $A = [2, 5, 3, 0, 2, 3, 0, 3]$, con $n = 8$ e $k = 5$.

Fase 1 – Inizializzazione. Si crea l'array C di dimensione $k + 1 = 6$:

$$C = [0, 0, 0, 0, 0, 0] \quad \text{(indici } 0, 1, 2, 3, 4, 5)$$

Fase 2 – Conteggio. Si scorre A da sinistra a destra, incrementando $C[A[j]]$ ad ogni passo:

j	A[j]	C dopo l'aggiornamento
1	2	$[0, 0, 1, 0, 0, 0]$
2	5	$[0, 0, 1, 0, 0, 1]$
3	3	$[0, 0, 1, 1, 0, 1]$
4	0	$[1, 0, 1, 1, 0, 1]$
5	2	$[1, 0, 2, 1, 0, 1]$
6	3	$[1, 0, 2, 2, 0, 1]$
7	0	$[2, 0, 2, 2, 0, 1]$
8	3	$[2, 0, 2, 3, 0, 1]$

Lettura di C: ci sono 2 zeri, 0 uni, 2 due, 3 tre, 0 quattro, 1 cinque.

Fase 3 – Somme prefisse. Si trasforma C in modo che $C[i]$ contenga il numero di elementi $\leq i$:

i	Calcolo	C
0	invariato	$[2, 0, 2, 3, 0, 1]$
1	$C[1] := 0 + 2 = 2$	$[2, 2, 2, 3, 0, 1]$
2	$C[2] := 2 + 2 = 4$	$[2, 2, 4, 3, 0, 1]$
3	$C[3] := 3 + 4 = 7$	$[2, 2, 4, 7, 0, 1]$
4	$C[4] := 0 + 7 = 7$	$[2, 2, 4, 7, 7, 1]$
5	$C[5] := 1 + 7 = 8$	$[2, 2, 4, 7, 7, 8]$

Interpretazione: ci sono 2 elementi ≤ 0, 2 elementi ≤ 1, 4 elementi ≤ 2, 7 elementi ≤ 3, 7 elementi ≤ 4, 8 elementi ≤ 5.

Fase 4 – Costruzione dell'output. Si scorre A **da destra a sinistra**. Per ogni $A[j]$, si inserisce $A[j]$ in posizione $C[A[j]]$ di B, poi si decrementa $C[A[j]]$:

j	A[j]	C[A[j]]	Posizionamento	B
8	3	7	$B[7] := 3$	$[-, -, -, -, -, -, 3, -]$
7	0	2	$B[2] := 0$	$[-, 0, -, -, -, -, 3, -]$
6	3	6	$B[6] := 3$	$[-, 0, -, -, -, 3, 3, -]$
5	2	4	$B[4] := 2$	$[-, 0, -, 2, -, 3, 3, -]$
4	0	1	$B[1] := 0$	$[0, 0, -, 2, -, 3, 3, -]$
3	3	5	$B[5] := 3$	$[0, 0, -, 2, 3, 3, 3, -]$
2	5	8	$B[8] := 5$	$[0, 0, -, 2, 3, 3, 3, 5]$
1	2	3	$B[3] := 2$	$[0, 0, 2, 2, 3, 3, 3, 5]$

Output finale: $B = [0, 0, 2, 2, 3, 3, 3, 5]$.

7.10.1.4 Stabilità

Definizione 7.9 – Algoritmo di ordinamento stabile. Un algoritmo di ordinamento è **stabile** se elementi con la stessa chiave di ordinamento mantengono l'ordine relativo che avevano nell'input originale.

Nota – Stabilità del Counting Sort. Il Counting Sort è **stabile** grazie al fatto che la Fase 4 scorre l'array A **da destra a sinistra**. Consideriamo due elementi $A[i]$ e $A[j]$ con $i < j$ e $A[i] = A[j]$. Poiché j viene elaborato prima di i (si parte da n e si decrementa), $A[j]$ viene posizionato in una posizione più alta (a destra) rispetto ad $A[i]$, preservando così l'ordine relativo originale.

Questa proprietà è fondamentale per il corretto funzionamento del Radix Sort, che richiede un sotto-algoritmo stabile per l'ordinamento su ogni cifra.

7.10.1.5 Complessità

Analizziamo separatamente ciascuna fase:

- **Fase 1** – Inizializzazione di C: $\Theta(k)$
- **Fase 2** – Conteggio degli elementi: $\Theta(n)$

- **Fase 3** – Calcolo delle somme prefisse: $\Theta(k)$
- **Fase 4** – Costruzione dell'output: $\Theta(n)$

La complessità totale è quindi:

$$T(n,k) = \Theta(n+k)$$

> **Nota – Quando il Counting Sort è lineare.** Se $k = O(n)$, cioè il range dei valori è proporzionale al numero di elementi, allora la complessità diventa $\Theta(n)$: ordinamento in tempo lineare. Tuttavia, se $k \gg n$ (ad esempio $k = n^2$), il Counting Sort diventa meno efficiente degli algoritmi basati su confronti.

7.10.1.6 Limitazioni

- Gli elementi devono essere **interi non negativi** (o mappabili a tali).
- Il range $[0, k]$ deve essere **noto a priori**.
- Lo **spazio ausiliario** richiesto è $\Theta(n+k)$ (per gli array B e C), quindi l'algoritmo **non è in-place**.
- Se k è molto grande rispetto a n, sia il tempo che lo spazio diventano proibitivi.

7.10.2 Radix Sort

Il **Radix Sort** ordina numeri interi (o stringhe) analizzando le singole cifre (o caratteri), dalla meno significativa (LSD, *Least Significant Digit*) alla più significativa (MSD, *Most Significant Digit*).

7.10.2.1 Idea

L'algoritmo esegue d passate sull'array, dove d è il numero di cifre del numero più lungo. Ad ogni passata, si ordina l'array rispetto a una singola cifra utilizzando un algoritmo di ordinamento **stabile** (tipicamente il Counting Sort). L'ordine di elaborazione delle cifre è cruciale: si parte dalla cifra meno significativa e si procede verso quella più significativa.

Perché dalla meno significativa? Se si procedesse dalla più significativa alla meno significativa, l'ordinamento sulle cifre successive potrebbe distruggere l'ordine già stabilito. Partendo dalla cifra meno significativa, la stabilità dell'algoritmo ausiliario garantisce che, quando si ordina per la cifra i, l'ordine relativo stabilito dalle cifre $1, ..., i-1$ viene preservato tra gli elementi con la stessa cifra i.

7.10.2.2 Algoritmo

Radix Sort

```
radixSort(int[] A, int d){
    // d = numero di cifre del valore massimo
    int i = 1;
    while(i <= d){
        // Ordina A con un algoritmo stabile sulla cifra i-esima
        stableSort(A, i);
        i := i + 1;
    }
}
```

7.10.2.3 Esempio passo-passo

Esempio 7.10 – Esecuzione di Radix Sort. Input: $A = [329, 457, 657, 839, 436, 720, 355]$, con $d = 3$ cifre (in base 10).

Passo 1 – Ordinamento per UNITÀ (cifra meno significativa).

Si estraggono le cifre delle unità e si ordina stabilmente rispetto ad esse:

Numero	Cifra delle unità
329	9
457	7
657	7
839	9
436	6
720	0
355	5

Risultato dopo ordinamento stabile per unità:

$$[720, 355, 436, 457, 657, 329, 839]$$

Passo 2 – Ordinamento per DECINE.

Si estraggono le cifre delle decine dall'array corrente e si ordina stabilmente:

Numero	Cifra delle decine
720	2
355	5
436	3
457	5
657	5
329	2
839	3

Risultato dopo ordinamento stabile per decine:

$$[720, 329, 436, 839, 355, 457, 657]$$

Si osservi la stabilità: 720 e 329 hanno entrambi cifra delle decine $= 2$, ma 720 precede 329 perché era già prima nell'array dopo il Passo 1. Analogamente, 355 precede 457 e 657 (tutti con decine $=$ 5), coerentemente con l'ordine del passo precedente.

Passo 3 – Ordinamento per CENTINAIA (cifra più significativa).

Si estraggono le cifre delle centinaia dall'array corrente e si ordina stabilmente:

Numero	Cifra delle centinaia
720	7
329	3
436	4
839	8
355	3
457	4
657	6

Risultato dopo ordinamento stabile per centinaia:

$$[329, 355, 436, 457, 657, 720, 839]$$

L'array è ora completamente ordinato. Si noti che 329 precede 355 (entrambi con centinaia $= 3$) perché dopo il Passo 2 il loro ordine relativo era già corretto ($29 < 55$) e la stabilità lo ha preservato.

7.10.2.4 Correttezza

Teorema 7.9 – Correttezza del Radix Sort. Se l'algoritmo ausiliario è stabile, dopo i iterazioni del Radix Sort gli elementi risultano ordinati rispetto alle ultime i cifre.

Dimostrazione. Per **induzione** sul numero di cifre i già elaborate.

Caso base ($i = 1$): dopo la prima iterazione, gli elementi sono ordinati rispetto alla cifra meno significativa (cifra 1), per correttezza dell'algoritmo stabile.

Passo induttivo: supponiamo che dopo $i - 1$ iterazioni gli elementi siano ordinati rispetto alle cifre $1, 2, ..., i - 1$ (ipotesi induttiva). Alla i-esima iterazione si ordina rispetto alla cifra i con un algoritmo stabile. Consideriamo due elementi x e y:

- Se la cifra i-esima di x è minore di quella di y, allora x precede y nell'output (per correttezza dell'ordinamento sulla cifra i).
- Se la cifra i-esima di x è maggiore di quella di y, allora y precede x.
- Se la cifra i-esima di x è uguale a quella di y, allora per la **stabilità** dell'algoritmo ausiliario, l'ordine relativo di x e y rimane invariato. Ma per ipotesi induttiva, questo ordine riflette correttamente l'ordinamento rispetto alle cifre $1, ..., i - 1$.

In tutti i casi, dopo i iterazioni gli elementi sono ordinati rispetto alle cifre $1, 2, ..., i$. Dopo d iterazioni l'array è completamente ordinato. ■

7.10.2.5 Complessità

Se si utilizza il Counting Sort come algoritmo stabile, e i numeri sono rappresentati in base b:

- Ogni cifra assume valori in $[0, b - 1]$, quindi il Counting Sort su ciascuna cifra richiede $\Theta(n + b)$.
- Il numero di cifre è: $d = \left\lceil \log_{b(M+1)} \right\rceil$, dove M è il valore massimo.

La complessità totale è:

$$T(n) = \Theta(d(n + b))$$

7.10.2.6 Scelta ottimale della base

> **Teorema 7.10 – Scelta ottimale della base per Radix Sort.** Dati n numeri interi rappresentabili con r bit (cioè con valori in $[0, 2^r - 1]$), e scegliendo la base $b = 2^s$ con $s = \lfloor \log_2 n \rfloor$ (quindi $b \approx n$), il Radix Sort ordina in tempo:
>
> $$T(n) = \Theta\left(\frac{r}{\log_2 n} \cdot n\right)$$
>
> Se $r = O(\log n)$ (cioè i valori sono polinomiali in n), la complessità diventa $\Theta(n)$: ordinamento lineare.

> **Nota – Intuizione sulla scelta della base.** Scegliere $b = n$ minimizza il prodotto $d \cdot (n + b)$:
> - Con una base troppo piccola (es. $b = 2$), si hanno molte cifre (d grande) e ogni passata è veloce ($n + 2$), ma il costo totale cresce.
> - Con una base troppo grande (es. $b = n^2$), si hanno poche cifre ma ogni passata richiede $\Theta(n + n^2)$.
> - Il bilanciamento ottimale si ottiene con $b \approx n$, che dà $d = r/\log n$ passate ciascuna di costo $\Theta(n)$.

7.10.3 Bucket Sort

Il **Bucket Sort** è un algoritmo di ordinamento che assume che l'input sia **distribuito uniformemente** in un intervallo noto, tipicamente $[0, 1)$.

7.10.3.1 Idea

L'algoritmo sfrutta la distribuzione uniforme dell'input per distribuire gli elementi in n **bucket** (contenitori) di uguale ampiezza. Poiché l'input è distribuito uniformemente, ci si aspetta che ogni bucket contenga pochi elementi, e quindi l'ordinamento interno a ciascun bucket sia rapido.

Le fasi dell'algoritmo sono:
1. **Distribuzione**: si dividono gli n elementi in n bucket. L'elemento $A[i]$ viene assegnato al bucket $\lfloor n \cdot A[i] \rfloor$.
2. **Ordinamento locale**: si ordina ciascun bucket con un algoritmo semplice (ad esempio Insertion Sort).
3. **Concatenazione**: si concatenano i bucket in ordine, ottenendo l'array ordinato.

7.10.3.2 Algoritmo

Bucket Sort

```
bucketSort(float[] A, int n){
    // Crea n liste (bucket) vuote
    List[] B = new List[n];

    // Distribuisci gli elementi nei bucket
    int i = 1;
    while(i <= n){
        int idx = floor(n * A[i]);
        insert(B[idx], A[i]);
        i := i + 1;
    }

    // Ordina ogni bucket con Insertion Sort
    i := 0;
    while(i <= n - 1){
        insertionSort(B[i]);
        i := i + 1;
    }
```

```
    // Concatena i bucket in ordine
    return concatenate(B[0], B[1], ..., B[n - 1]);
}
```

Nota. Si assume che gli elementi siano numeri reali in $[0, 1)$. Per input in un intervallo arbitrario $[a, b)$, si normalizza: $A[i] := \frac{A[i]-a}{b-a}$.

7.10.3.3 Esempio passo-passo

Esempio 7.11 – Esecuzione di Bucket Sort. Input: $A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]$, con $n = 10$.

Fase 1 – Distribuzione. Ogni elemento $A[i]$ viene assegnato al bucket di indice $\lfloor 10 \cdot A[i] \rfloor$:

Elemento	$\lfloor 10 \cdot A[i] \rfloor$	**Bucket**
0.78	7	$B[7]$
0.17	1	$B[1]$
0.39	3	$B[3]$
0.26	2	$B[2]$
0.72	7	$B[7]$
0.94	9	$B[9]$
0.21	2	$B[2]$
0.12	1	$B[1]$
0.23	2	$B[2]$
0.68	6	$B[6]$

Contenuto dei bucket dopo la distribuzione:
- $B[1] = \langle 0.17, 0.12 \rangle$
- $B[2] = \langle 0.26, 0.21, 0.23 \rangle$
- $B[3] = \langle 0.39 \rangle$
- $B[6] = \langle 0.68 \rangle$
- $B[7] = \langle 0.78, 0.72 \rangle$
- $B[9] = \langle 0.94 \rangle$
- (gli altri bucket sono vuoti)

Fase 2 – Ordinamento locale. Si ordina ogni bucket con Insertion Sort:
- $B[1] = \langle 0.12, 0.17 \rangle$
- $B[2] = \langle 0.21, 0.23, 0.26 \rangle$
- $B[3] = \langle 0.39 \rangle$
- $B[6] = \langle 0.68 \rangle$
- $B[7] = \langle 0.72, 0.78 \rangle$
- $B[9] = \langle 0.94 \rangle$

Fase 3 – Concatenazione.

$$B = [0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]$$

7.10.3.4 Complessità

Teorema 7.11 – Complessità del Bucket Sort.
- **Caso pessimo**: $\Theta(n^2)$. Si verifica quando tutti gli elementi finiscono nello stesso bucket, riducendo l'algoritmo a un Insertion Sort sull'intero array.
- **Caso medio** (con input distribuito uniformemente in $[0, 1)$): $\Theta(n)$.

Dimostrazione. Sia n_i il numero di elementi nel bucket i. Il tempo totale è:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

dove $\Theta(n)$ è il costo della distribuzione e della concatenazione, e $O(n_i^2)$ è il costo dell'Insertion Sort sul bucket i.

Calcoliamo il valore atteso. Se l'input è distribuito uniformemente in $[0, 1)$, ogni elemento finisce nel bucket i con probabilità $1/n$. Quindi n_i segue una distribuzione binomiale $B(n, 1/n)$, con $E[n_i] = 1$ e $\mathrm{Var}(n_i) = 1 - 1/n$.

$$E[n_i^2] = \mathrm{Var}(n_i) + (E[n_i])^2 = \left(1 - \frac{1}{n}\right) + 1 = 2 - \frac{1}{n}$$

Pertanto:

$$E[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right) = \Theta(n) + O(n) = \Theta(n)$$

■

7.10.3.5 Limitazioni

- Richiede l'ipotesi di **distribuzione uniforme** dell'input per garantire complessità media lineare.
- Le prestazioni degradano significativamente con distribuzioni non uniformi (molti elementi nello stesso bucket).
- Richiede **spazio ausiliario** $\Theta(n)$ per i bucket.
- L'algoritmo **non è in-place**.

7.10.4 Riepilogo degli algoritmi di ordinamento

La tabella seguente riassume le caratteristiche di tutti gli algoritmi di ordinamento studiati.

Algoritmo	Caso pessimo	Caso medio	Caso migliore	Stabile?	In-place?
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Sì	Sì
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	No	Sì
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	Sì	No
QuickSort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	No	Sì
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	No	Sì
Counting Sort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$	Sì	No
Radix Sort	$\Theta(d(n + b))$	$\Theta(d(n + b))$	$\Theta(d(n + b))$	Sì	No
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$	Sì	No

Tabella 19: Confronto completo degli algoritmi di ordinamento

Nota – Guida alla lettura della tabella.
- $\boldsymbol{k}$: range dei valori nel Counting Sort ($[0, k]$).
- $\boldsymbol{d}$: numero di cifre nel Radix Sort.
- $\boldsymbol{b}$: base di rappresentazione nel Radix Sort (ciascuna cifra in $[0, b-1]$).
- **Stabile**: un algoritmo è stabile se preserva l'ordine relativo degli elementi con chiave uguale.
- **In-place**: un algoritmo è in-place se utilizza solo $O(1)$ spazio ausiliario (oltre all'input).

Nota – Osservazioni finali.
- Gli algoritmi **basati su confronti** (Insertion Sort, Selection Sort, MergeSort, QuickSort, HeapSort) hanno un limite inferiore di $\Omega(n \log n)$ nel caso pessimo. MergeSort e HeapSort raggiungono questo limite e sono quindi asintoticamente ottimi.
- **Counting Sort, Radix Sort e Bucket Sort** possono raggiungere complessità lineare, ma richiedono ipotesi aggiuntive sull'input: interi in un range limitato per i primi due, distribuzione uniforme per il terzo.
- **QuickSort**, nonostante il caso pessimo $\Theta(n^2)$, è spesso preferito nella pratica per il suo eccellente caso medio e per le costanti moltiplicative basse.
- La scelta dell'algoritmo dipende dalle **caratteristiche dei dati** (tipo, distribuzione, range), dai **vincoli di spazio**, e dalla necessità di **stabilità**.

8 Strutture Dati

8.1 Strutture Dati Lineari

Una **struttura dati** è un modo sistematico di organizzare e memorizzare dati in modo da rendere efficienti le operazioni che si intendono eseguire su di essi. La scelta della struttura dati influenza direttamente la complessità temporale e spaziale degli algoritmi.

Le **strutture dati lineari** organizzano gli elementi in sequenza: ogni elemento (tranne il primo e l'ultimo) ha esattamente un predecessore e un successore. Le strutture che analizziamo in questa sezione sono **array**, **liste concatenate**, **pile** e **code**.

8.1.1 Array

> **Definizione 8.1 – Array.** Un **array** è una struttura dati che memorizza una collezione di n elementi dello stesso tipo in n locazioni di memoria **contigue**. Ogni elemento è univocamente identificato da un **indice** intero $i \in \{1, ..., n\}$ e si accede ad esso in tempo costante tramite la formula:
>
> $$\text{indirizzo}(A[i]) = \text{indirizzo}(A[1]) + (i-1) \cdot \text{dimensione elemento}$$

L'accesso diretto tramite indice e la contiguita in memoria sono le proprieta fondamentali degli array. La contiguita garantisce un'elevata **localita spaziale**, che si traduce in un utilizzo efficiente della cache del processore: quando si accede a $A[i]$, gli elementi vicini $A[i+1], A[i+2], ...$ vengono caricati nella stessa linea di cache.

8.1.1.1 Proprieta degli array

- **Accesso diretto** in $O(1)$ tramite indice
- **Dimensione fissa**: stabilita al momento della creazione (in molti linguaggi)
- **Elementi contigui in memoria**: ottimo per accessi sequenziali
- **Tipo omogeneo**: tutti gli elementi hanno lo stesso tipo

8.1.1.2 Operazioni e complessita

Operazione	Descrizione	Complessita
`Access(A, i)`	Restituisce l'elemento in posizione i	$O(1)$
`Search(A, x)`	Cerca l'elemento x (scansione lineare)	$O(n)$
`Insert(A, i, x)`	Inserisce x in posizione i, traslando gli elementi successivi	$O(n)$
`Delete(A, i)`	Elimina l'elemento in posizione i, traslando gli elementi successivi	$O(n)$

Tabella 20: Operazioni su array e relative complessita nel caso peggiore

> **Nota – Costo dell'inserimento e della cancellazione.** L'inserimento di un elemento in posizione i richiede lo spostamento di tutti gli elementi da $A[i]$ a $A[n]$ di una posizione a destra, per fare spazio al nuovo elemento. Analogamente, la cancellazione richiede lo spostamento degli elementi successivi a sinistra per riempire il «buco». Nel caso peggiore (inserimento o cancellazione in testa, $i = 1$) si spostano tutti gli n elementi, da cui la complessita $O(n)$.
>
> L'inserimento **in coda** (posizione $n+1$) costa $O(1)$ se c'e spazio disponibile, perche non richiede alcuno spostamento.

Esempio 8.1 – Inserimento in un array. Dato l'array $A = [3, 7, 12, 5, 9]$ con $n = 5$, inseriamo il valore 8 in posizione $i = 3$:

1. Spostiamo $A[5]$ in $A[6]$: $[3, 7, 12, 5, 9, 9]$
2. Spostiamo $A[4]$ in $A[5]$: $[3, 7, 12, 5, 5, 9]$
3. Spostiamo $A[3]$ in $A[4]$: $[3, 7, 12, 12, 5, 9]$
4. Scriviamo $A[3] := 8$: $[3, 7, 8, 12, 5, 9]$

Sono stati necessari $n - i + 1 = 3$ spostamenti.

8.1.2 Liste Concatenate

Definizione 8.2 – Lista concatenata. Una **lista concatenata** (linked list) e una struttura dati in cui gli elementi, detti **nodi**, sono collegati tramite **puntatori**. A differenza degli array, i nodi non occupano locazioni di memoria contigue: ogni nodo puo trovarsi in una posizione arbitraria della memoria, e il collegamento tra nodi e realizzato esclusivamente attraverso i puntatori.

Il vantaggio principale delle liste rispetto agli array e che inserimenti e cancellazioni possono essere effettuati in tempo $O(1)$ senza spostare altri elementi, purche si disponga del puntatore al punto di inserimento. Lo svantaggio e la perdita dell'accesso diretto: per raggiungere l'i-esimo elemento bisogna attraversare $i - 1$ nodi.

8.1.2.1 Lista semplicemente concatenata

In una lista **semplicemente concatenata** ogni nodo contiene due campi:

- `key`: il dato memorizzato nel nodo
- `next`: un puntatore al nodo successivo nella lista (oppure `NIL` se il nodo e l'ultimo)

La lista e accessibile tramite un puntatore `L.head` al primo nodo.

Struttura di un nodo (lista semplice)

```
// Nodo di lista semplicemente concatenata
// key  : dato memorizzato
// next : puntatore al nodo successivo (NIL se ultimo)
```

Esempio 8.2 – Lista semplicemente concatenata. La lista $\langle 3, 7, 12, 5 \rangle$ e rappresentata come:

$$\underbrace{[3 \mid \cdot] \to [7 \mid \cdot] \to [12 \mid \cdot] \to [5 \mid /]}_{\text{L.head}}$$

Il simbolo / indica il puntatore `NIL`. Ogni nodo contiene il dato e un puntatore al nodo successivo.

8.1.2.2 Lista doppiamente concatenata

In una lista **doppiamente concatenata** ogni nodo contiene tre campi:

- `key`: il dato memorizzato
- `next`: puntatore al nodo successivo
- `prev`: puntatore al nodo precedente

Struttura di un nodo (lista doppia)

```
// Nodo di lista doppiamente concatenata
// key  : dato memorizzato
// next : puntatore al nodo successivo (NIL se ultimo)
// prev : puntatore al nodo precedente (NIL se primo)
```

La presenza del puntatore `prev` consente la navigazione in entrambe le direzioni e semplifica l'operazione di cancellazione: dato un puntatore a un nodo x, si puo rimuoverlo in $O(1)$ senza dover cercare il suo predecessore.

8.1.2.3 Operazioni su liste doppiamente concatenate

Le tre operazioni fondamentali su liste sono la **ricerca**, l'**inserimento** e la **cancellazione**. Presentiamo lo pseudocodice per una lista doppiamente concatenata con puntatore `L.head` alla testa.

8.1.2.3.1 Ricerca

La ricerca scorre la lista dal primo nodo fino a trovare un nodo con chiave k oppure fino a raggiungere la fine della lista (`NIL`).

List-Search(L, k)

```
Node listSearch(List L, int k){
    Node x = L.head;
    while((x != NIL) && (x.key != k)){
        x := x.next;
    }
    return x;
}
```

Complessita: $O(n)$ nel caso peggiore (elemento non presente o in ultima posizione). Nel caso migliore $O(1)$ (elemento in testa).

8.1.2.3.2 Inserimento in testa

L'inserimento in testa aggiunge un nuovo nodo x come primo elemento della lista.

List-Insert(L, x)

```
listInsert(List L, Node x){
    x.next := L.head;
    if(L.head != NIL){
        L.head.prev := x;
    }
    L.head := x;
    x.prev := NIL;
}
```

Complessita: $O(1)$, poiche si aggiornano solo un numero costante di puntatori indipendentemente dalla lunghezza della lista.

8.1.2.3.3 Cancellazione

La cancellazione rimuove un nodo x dalla lista, dato il puntatore a x. Si devono aggiornare i puntatori del predecessore e del successore di x affinche si «scavalchi» il nodo rimosso.

List-Delete(L, x)

```
listDelete(List L, Node x){
    if(x.prev != NIL){
        x.prev.next := x.next;
    } else {
        L.head := x.next;
    }
    if(x.next != NIL){
        x.next.prev := x.prev;
    }
}
```

Complessita: $O(1)$ se si dispone gia del puntatore al nodo x. Se occorre prima cercare x (ad esempio tramite la chiave), la complessita complessiva diventa $O(n)$ a causa della ricerca.

> **Nota – Gestione dei casi limite.** Nell'operazione `List-Delete`, il ramo `else` gestisce il caso in cui x e il primo nodo della lista (`x.prev == NIL`): in tal caso occorre aggiornare `L.head`. Analogamente, il secondo `if` gestisce il caso in cui x e l'ultimo nodo. Questa gestione dei casi limite rende il codice piu complesso; le **sentinelle** permettono di semplificarlo.

8.1.2.4 Varianti delle liste

8.1.2.4.1 Lista circolare

In una **lista circolare**, l'ultimo nodo punta al primo invece che a `NIL`. In una lista doppiamente concatenata circolare, il `prev` del primo nodo punta all'ultimo e il `next` dell'ultimo punta al primo. Questa struttura e utile quando si vuole poter ciclare sugli elementi senza un punto di inizio/fine esplicito.

8.1.2.4.2 Lista con sentinella

Una **sentinella** (o nodo fittizio) e un nodo speciale `L.nil` che non contiene dati significativi e funge da «confine» della lista. In una lista doppiamente concatenata con sentinella:

- `L.nil.next` punta al primo elemento della lista (la «testa»)
- `L.nil.prev` punta all'ultimo elemento (la «coda»)
- Il `prev` del primo elemento punta a `L.nil`
- Il `next` dell'ultimo elemento punta a `L.nil`
- Una lista vuota ha `L.nil.next == L.nil` e `L.nil.prev == L.nil`

La sentinella trasforma la lista in una struttura circolare e **elimina tutti i casi limite** (lista vuota, inserimento/cancellazione in testa o in coda), semplificando notevolmente lo pseudocodice.

Cancellazione con sentinella

```
// Con sentinella: nessun caso speciale necessario
listDelete'(List L, Node x){
    x.prev.next := x.next;
    x.next.prev := x.prev;
}
```

> **Nota – Quando usare le sentinelle.** Le sentinelle semplificano il codice ma non migliorano la complessita asintotica. Sono utili quando si hanno molte operazioni di inserimento e cancellazione e si vuole ridurre il numero di condizioni da verificare. Per pochi elementi, il nodo sentinella aggiuntivo rappresenta un overhead di memoria non trascurabile.

8.1.2.5 Confronto Array vs Liste

Operazione	**Array**	**Lista concatenata**
Accesso per indice	$O(1)$	$O(n)$
Ricerca (non ordinato)	$O(n)$	$O(n)$
Inserimento in testa	$O(n)$	$O(1)$
Inserimento in coda	$O(1)^*$	$O(1)^{**}$
Cancellazione (dato il puntatore/indice)	$O(n)$	$O(1)$
Uso della memoria	Compatto, no overhead	Overhead per puntatori
Localita di cache	Elevata	Scarsa

Tabella 21: Confronto tra array e lista concatenata. * se c'e spazio nell'array. ** se si mantiene un puntatore alla coda.

La scelta tra array e lista dipende dal profilo di utilizzo: se predominano accessi per indice, l'array e preferibile; se predominano inserimenti e cancellazioni in posizioni arbitrarie, la lista concatenata e piu efficiente.

8.1.3 Pile (Stack)

> **Definizione 8.3 – Pila.** Una **pila** (stack) e una struttura dati con politica di accesso **LIFO** (Last In, First Out): l'ultimo elemento inserito e il primo ad essere rimosso. L'unico elemento accessibile e quello in **cima** (top) alla pila.

La pila e un **tipo di dato astratto**: la definizione specifica **cosa** fa (l'interfaccia), non **come** e implementata. Puo essere realizzata sia con array che con liste concatenate.

8.1.3.1 Interfaccia

Le operazioni fondamentali su una pila S sono:

- `Push(S, x)`: inserisce l'elemento x in cima alla pila
- `Pop(S)`: rimuove e restituisce l'elemento in cima alla pila. Errore (**underflow**) se la pila e vuota
- `Top(S)`: restituisce l'elemento in cima senza rimuoverlo
- `IsEmpty(S)`: restituisce `true` se la pila e vuota, `false` altrimenti

Tutte le operazioni hanno complessita $O(1)$.

8.1.3.2 Implementazione con array

Si utilizza un array $S[1..n]$ e una variabile `S.top` che indica l'indice dell'elemento in cima. Una pila vuota ha `S.top == 0`.

Pila su array

```
bool stackEmpty(Stack S){
    return S.top == 0;
}

push(Stack S, int x){
    if(S.top == S.length){
        // error "overflow"
        return;
    }
    S.top := S.top + 1;
    S[S.top] := x;
}

int pop(Stack S){
    if(stackEmpty(S)){
        // error "underflow"
        return -1;
    }
    S.top := S.top - 1;
    return S[S.top + 1];
}

int top(Stack S){
    if(stackEmpty(S)){
        // error "underflow"
        return -1;
    }
    return S[S.top];
}
```

> **Nota – Overflow.** Nell'implementazione su array, la dimensione della pila e limitata dalla dimensione dell'array. Se si tenta un `Push` quando `S.top == S.length`, si verifica un errore di **overflow**. Questo problema non si presenta nell'implementazione su lista.

8.1.3.3 Implementazione con lista concatenata

La cima della pila corrisponde alla testa della lista. Le operazioni `Push` e `Pop` corrispondono rispettivamente all'inserimento e alla cancellazione in testa.

Pila su lista concatenata

```
push(Stack S, Node x){
    x.next := S.head;
    S.head := x;
}

Node pop(Stack S){
    if(S.head == NIL){
        // error "underflow"
        return NIL;
    }
    Node x = S.head;
    S.head := S.head.next;
    return x;
}
```

> **Nota – Confronto delle due implementazioni.**

- **Array**: piu efficiente in termini di memoria (nessun overhead per puntatori) e migliore localita di cache, ma ha dimensione fissa.
- **Lista**: dimensione dinamica (nessun overflow), ma ogni nodo richiede spazio aggiuntivo per il puntatore `next`.

8.1.3.4 Applicazioni delle pile

- **Call stack**: gestione delle chiamate di funzione e dei record di attivazione. Quando una funzione f chiama una funzione g, il contesto di f viene salvato sulla pila; al ritorno di g, viene ripristinato.
- **Valutazione di espressioni**: conversione e valutazione di espressioni in notazione polacca inversa (postfissa).
- **Algoritmi di backtracking**: esplorazione di spazi di ricerca (es. labirinti, puzzle) dove si «torna indietro» all'ultimo punto di scelta.
- **Undo/Redo**: negli editor di testo, ogni azione viene impilata; l'annullamento corrisponde a un `Pop`.
- **Bilanciamento delle parentesi**: verifica che ogni parentesi aperta abbia la corrispondente chiusa.

Esempio 8.3 – Bilanciamento delle parentesi. Per verificare se una stringa di parentesi e correttamente bilanciata, si utilizza una pila: ogni parentesi aperta viene impilata, e per ogni parentesi chiusa si verifica che corrisponda alla parentesi in cima alla pila.

isBalanced(s, n)

```
bool isBalanced(char[] s, int n){
    Stack S = new Stack();
    int i = 0;
    while(i < n){
        char c = s[i];
        if((c == '(') || (c == '[') || (c == '{')){
            push(S, c);
        } else {
            if((c == ')') || (c == ']') || (c == '}')){
                if(stackEmpty(S)){
                    return false;
                }
                char open = pop(S);
                if(!match(open, c)){
                    return false;
                }
            }
        }
        i := i + 1;
    }
    return stackEmpty(S);
}
```

Ad esempio, sulla stringa `"({[]})"`:
1. Leggiamo `(`: impiliamo → pila: $\langle ($
2. Leggiamo `{`: impiliamo → pila: $\langle (, \{$
3. Leggiamo `[`: impiliamo → pila: $\langle (, \{, [$
4. Leggiamo `]`: `Pop` restituisce `[`, che corrisponde → pila: $\langle (, \{$
5. Leggiamo `}`: `Pop` restituisce `{`, che corrisponde → pila: $\langle ($
6. Leggiamo `)`: `Pop` restituisce `(`, che corrisponde → pila: $\langle\rangle$ (vuota)

7. Stringa terminata e pila vuota → **bilanciata**.

Esempio 8.4 – Esecuzione di Push e Pop. Mostriamo l'evoluzione della pila durante una sequenza di operazioni, usando l'implementazione su array $S[1..6]$ con `S.top` inizialmente a 0:

Operazione	S.top	Contenuto array	Cima
`Push(S, 4)`	1	$[4, -, -, -, -, -]$	4
`Push(S, 7)`	2	$[4, 7, -, -, -, -]$	7
`Push(S, 2)`	3	$[4, 7, 2, -, -, -]$	2
`Pop(S)` $\to 2$	2	$[4, 7, -, -, -, -]$	7
`Push(S, 9)`	3	$[4, 7, 9, -, -, -]$	9
`Pop(S)` $\to 9$	2	$[4, 7, -, -, -, -]$	7
`Pop(S)` $\to 7$	1	$[4, -, -, -, -, -]$	4

Tabella 22: Evoluzione della pila durante operazioni Push e Pop

8.1.4 Code (Queue)

Definizione 8.4 – Coda. Una **coda** (queue) e una struttura dati con politica di accesso **FIFO** (First In, First Out): il primo elemento inserito e il primo ad essere rimosso. Gli inserimenti avvengono alla **fine** (tail) e le rimozioni all'**inizio** (head) della coda.

8.1.4.1 Interfaccia

Le operazioni fondamentali su una coda Q sono:

- `Enqueue(Q, x)`: inserisce l'elemento x alla fine della coda
- `Dequeue(Q)`: rimuove e restituisce l'elemento all'inizio della coda. Errore (**underflow**) se la coda e vuota
- `Front(Q)`: restituisce l'elemento all'inizio senza rimuoverlo
- `IsEmpty(Q)`: restituisce `true` se la coda e vuota, `false` altrimenti

Tutte le operazioni hanno complessita $O(1)$.

8.1.4.2 Implementazione con array circolare

Un'implementazione ingenua su array (inserimento in coda e rimozione in testa) porterebbe a uno spostamento progressivo degli elementi verso destra, sprecando spazio. La soluzione e l'**array circolare**: si utilizza un array $Q[1..n]$ in cui gli indici «ruotano», tornando alla posizione 1 dopo la posizione n.

Si mantengono due indici:
- `Q.head`: posizione del primo elemento (prossimo da rimuovere)
- `Q.tail`: posizione della prossima cella libera (dove inserire il prossimo elemento)

La coda e vuota quando `Q.head == Q.tail`. L'operazione di «avanzamento circolare» dell'indice si esprime come:

$$i := (i \bmod n) + 1$$

Questo garantisce che dopo la posizione n si torni alla posizione 1.

Coda su array circolare

```
enqueue(Queue Q, int x){
    if(((Q.tail mod Q.length) + 1) == Q.head){
        // error "overflow"
        return;
    }
    Q[Q.tail] := x;
    if(Q.tail == Q.length){
        Q.tail := 1;
    } else {
        Q.tail := Q.tail + 1;
    }
}

int dequeue(Queue Q){
    if(Q.head == Q.tail){
        // error "underflow"
        return -1;
    }
    int x = Q[Q.head];
    if(Q.head == Q.length){
        Q.head := 1;
    } else {
        Q.head := Q.head + 1;
    }
    return x;
}
```

Nota – Capacita effettiva. Con questa implementazione, un array di dimensione n puo contenere al massimo $n-1$ elementi. Si «sacrifica» una posizione per poter distinguere la coda vuota (`Q.head == Q.tail`) dalla coda piena (`(Q.tail mod n) + 1 == Q.head`). Un'alternativa e mantenere un contatore esplicito del numero di elementi.

Esempio 8.5 – Esecuzione su array circolare. Array $Q[1..5]$, inizialmente `Q.head = Q.tail = 1` (coda vuota):

Operazione	head	tail	Contenuto
(iniziale)	1	1	$[-,-,-,-,-]$
`Enqueue(Q, 3)`	1	2	$[3,-,-,-,-]$
`Enqueue(Q, 7)`	1	3	$[3,7,-,-,-]$
`Enqueue(Q, 5)`	1	4	$[3,7,5,-,-]$
`Dequeue(Q)` $\to 3$	2	4	$[-,7,5,-,-]$
`Enqueue(Q, 12)`	2	5	$[-,7,5,12,-]$
`Enqueue(Q, 1)`	2	1	$[1,7,5,12,-]$
`Dequeue(Q)` $\to 7$	3	1	$[1,-,5,12,-]$

Tabella 23: Evoluzione della coda su array circolare. Si noti come il tail «ritorna» alla posizione 1 dopo la posizione 5.

8.1.4.3 Implementazione con lista concatenata

Si mantengono due puntatori: `Q.head` al primo elemento e `Q.tail` all'ultimo. L'inserimento avviene in coda (tramite `Q.tail`) e la rimozione in testa (tramite `Q.head`).

Coda su lista concatenata

```
enqueue(Queue Q, Node x){
    x.next := NIL;
    if(Q.tail != NIL){
        Q.tail.next := x;
    }
    Q.tail := x;
    if(Q.head == NIL){
        Q.head := x;
    }
}

Node dequeue(Queue Q){
    if(Q.head == NIL){
        // error "underflow"
        return NIL;
    }
    Node x = Q.head;
    Q.head := Q.head.next;
    if(Q.head == NIL){
        Q.tail := NIL;
    }
    return x;
}
```

> **Nota – Correttezza della coda su lista.** Nell'operazione `Enqueue`: se la coda e vuota (`Q.head == NIL`), il nuovo nodo diventa sia la testa che la coda. Altrimenti, si aggiunge il nodo dopo l'attuale coda.
>
> Nell'operazione `Dequeue`: se dopo la rimozione `Q.head` diventa `NIL`, anche `Q.tail` deve essere posto a `NIL` (la coda e diventata vuota).

8.1.4.4 Applicazioni delle code

- **Scheduling di processi**: il sistema operativo gestisce i processi pronti per l'esecuzione in una coda FIFO (round-robin scheduling).
- **Buffer di I/O**: i dati in ingresso e in uscita vengono bufferizzati in code per gestire la differenza di velocita tra produttore e consumatore.
- **Gestione delle richieste**: i web server utilizzano code per servire le richieste dei client nell'ordine di arrivo.
- **BFS (Breadth-First Search)**: l'algoritmo di visita in ampiezza dei grafi utilizza una coda per esplorare i nodi livello per livello.

8.1.5 Deque (Double-Ended Queue)

> **Definizione 8.5 – Deque.** Una **deque** (double-ended queue, pronunciato «deck») e una struttura dati che generalizza sia la pila che la coda, permettendo inserimenti e rimozioni da **entrambe** le estremita.

8.1.5.1 Interfaccia

- `InsertFront(D, x)`: inserisce x in testa
- `InsertBack(D, x)`: inserisce x in coda
- `RemoveFront(D)`: rimuove e restituisce l'elemento in testa
- `RemoveBack(D)`: rimuove e restituisce l'elemento in coda

Tutte le operazioni hanno complessita $O(1)$.

> **Nota.** Una deque puo simulare sia una pila (usando solo `InsertFront` e `RemoveFront`) che una coda (usando `InsertBack` e `RemoveFront`). Puo essere implementata efficientemente con una lista doppiamente concatenata oppure con un array circolare con due indici.

8.2 Code con priorità

Le **code con priorità** sono trattate nella sezione dedicata agli Heap nel capitolo Algoritmi di Ordinamento. A differenza delle code FIFO, l'elemento rimosso non è il primo inserito ma quello con **priorità massima** (o minima). Le operazioni `Insert` e `Extract-Max` (o `Extract-Min`) hanno complessità $O(\log n)$ quando implementate con un heap binario.

8.3 Riepilogo e confronto

Struttura	**Politica**	**Inserimento**	**Rimozione**	**Accesso**
Array	Indice	$O(n)$	$O(n)$	$O(1)$
Lista	Posizione	$O(1)^*$	$O(1)^*$	$O(n)$
Pila	LIFO	$O(1)$	$O(1)$	$O(1)^{**}$
Coda	FIFO	$O(1)$	$O(1)$	$O(1)^{**}$
Deque	Doppia	$O(1)$	$O(1)$	$O(1)^{**}$

Tabella 24: Riepilogo delle complessita. * Con puntatore alla posizione. ** Solo all'elemento accessibile (cima o fronte).

> **Nota – Criteri di scelta.** La scelta della struttura dati dipende dal tipo di operazioni che l'algoritmo deve eseguire con maggiore frequenza:
> - **Accesso casuale frequente** → Array
> - **Inserimenti e cancellazioni frequenti in posizioni arbitrarie** → Lista concatenata
> - **Elaborazione LIFO** (ultimo arrivato, primo servito) → Pila
> - **Elaborazione FIFO** (primo arrivato, primo servito) → Coda
> - **Inserimenti/rimozioni da entrambe le estremita** → Deque

8.4 Alberi Binari

8.4.1 Definizione e terminologia

> **Definizione 8.6 – Albero binario.** Un **albero binario** è una struttura dati definita ricorsivamente come:
> - l'**albero vuoto** (indicato con NIL), oppure
> - un **nodo** x che contiene una chiave x.key e due puntatori a sottoalberi binari: il **figlio sinistro** x.left e il **figlio destro** x.right.

> **Definizione 8.7 – Terminologia.** Dato un albero binario T:
> - **Radice**: l'unico nodo senza padre, indicato con T.root
> - **Foglia**: un nodo con entrambi i figli uguali a NIL

- **Nodo interno**: un nodo con almeno un figlio diverso da NIL
- **Profondità** di un nodo: il numero di archi dal nodo alla radice (la radice ha profondità 0)
- **Altezza** di un nodo: il numero di archi nel cammino più lungo dal nodo a una foglia
- **Altezza dell'albero**: l'altezza della radice, indicata con h
- **Livello** k: l'insieme dei nodi a profondità k
- **Padre** di un nodo x: il nodo y tale che $y.\text{left} = x$ oppure $y.\text{right} = x$

8.4.2 Rappresentazione di un nodo

Ogni nodo di un albero binario contiene:

- `key`: il dato memorizzato
- `left`: puntatore al figlio sinistro (oppure NIL)
- `right`: puntatore al figlio destro (oppure NIL)
- `parent`: puntatore al nodo padre (oppure NIL per la radice)

Struttura Nodo

```
// Struttura Nodo (albero binario)
// key    : dato
// left   : puntatore al figlio sinistro
// right  : puntatore al figlio destro
// parent : puntatore al padre
```

8.4.3 Proprietà degli alberi binari

Teorema 8.1 – Relazione nodi-altezza. Sia T un albero binario di altezza h. Allora:

- T ha al massimo $2^{h+1} - 1$ nodi
- T ha al massimo 2^h foglie
- Se T ha n nodi, allora $h \geq \lceil \log_2(n+1) \rceil - 1$

Dimostrazione. Per induzione sull'altezza h.

Caso base ($h = 0$): l'albero ha un solo nodo (la radice, che è anche foglia). Si ha $2^{0+1} - 1 = 1$ nodo e $2^0 = 1$ foglia.

Passo induttivo: supponiamo la tesi vera per altezza $h - 1$. Un albero di altezza h ha una radice con due sottoalberi di altezza al più $h - 1$. Per ipotesi induttiva, ciascun sottoalbero ha al più $2^h - 1$ nodi, quindi il totale è al più $1 + 2\big(2^h - 1\big) = 2^{h+1} - 1$.

Per la terza proprietà: da $n \leq 2^{h+1} - 1$ si ottiene $h \geq \lceil \log_2(n+1) \rceil - 1$. ■

Osservazione. Un albero binario **completo** (ogni nodo interno ha esattamente due figli e tutte le foglie sono allo stesso livello) di altezza h ha esattamente $2^{h+1} - 1$ nodi. Un albero binario **degenere** (ogni nodo interno ha esattamente un figlio) di altezza h ha esattamente $h + 1$ nodi ed è equivalente a una lista concatenata.

8.4.4 Visite di un albero binario

Le visite (o attraversamenti) di un albero binario permettono di esaminare tutti i nodi in un ordine specifico. Le tre visite fondamentali si distinguono per il momento in cui viene visitata la radice rispetto ai sottoalberi.

8.4.4.1 Visita in ordine (inorder)

La visita **inorder** visita prima il sottoalbero sinistro, poi la radice, poi il sottoalbero destro.

inorderTreeWalk

```
inorderTreeWalk(Node x){
    if(x != NIL){
        inorderTreeWalk(x.left);
        // visita x (es. stampa x.key)
        inorderTreeWalk(x.right);
    }
}
```

8.4.4.2 Visita anticipata (preorder)

La visita **preorder** visita prima la radice, poi il sottoalbero sinistro, poi il destro.

preorderTreeWalk

```
preorderTreeWalk(Node x){
    if(x != NIL){
        // visita x (es. stampa x.key)
        preorderTreeWalk(x.left);
        preorderTreeWalk(x.right);
    }
}
```

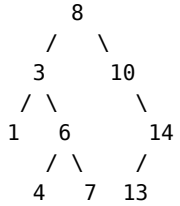
8.4.4.3 Visita posticipata (postorder)

La visita **postorder** visita prima il sottoalbero sinistro, poi il destro, poi la radice.

postorderTreeWalk

```
postorderTreeWalk(Node x){
    if(x != NIL){
        postorderTreeWalk(x.left);
        postorderTreeWalk(x.right);
        // visita x (es. stampa x.key)
    }
}
```

Esempio 8.6 – Visite di un albero. Consideriamo il seguente albero binario:



Le tre visite producono:

- **Inorder**: $1, 3, 4, 6, 7, 8, 10, 13, 14$
- **Preorder**: $8, 3, 1, 6, 4, 7, 10, 14, 13$
- **Postorder**: $1, 4, 7, 6, 3, 13, 14, 10, 8$

Teorema 8.2 – Complessità delle visite. Ciascuna delle tre visite di un albero binario con n nodi ha complessità $\Theta(n)$.

Dimostrazione. Sia $T(n)$ il tempo per visitare un albero con n nodi. Se l'albero è vuoto, $T(0) = c$ per una costante $c > 0$. Altrimenti, se il sottoalbero sinistro ha k nodi e il destro ne ha $n - k - 1$:

$$T(n) = T(k) + T(n - k - 1) + d$$

dove d è il costo costante per visitare il nodo corrente. Per sostituzione si dimostra che $T(n) = (c + d)n + c = \Theta(n)$. ■

8.5 Alberi Binari di Ricerca (BST)

8.5.1 Proprietà BST

Definizione 8.8 – Albero binario di ricerca. Un **albero binario di ricerca** (BST, Binary Search Tree) è un albero binario in cui per ogni nodo x vale la seguente proprietà:

- per ogni nodo y nel sottoalbero sinistro di x: y.key $< x$.key
- per ogni nodo y nel sottoalbero destro di x: y.key $\geq x$.key

In altre parole, le chiavi strettamente minori si trovano nel sottoalbero sinistro, mentre le chiavi uguali o maggiori si trovano nel sottoalbero destro.

Nota. La proprietà BST è una proprietà **globale**: non basta che ogni nodo sia maggiore del figlio sinistro e minore del figlio destro; occorre che la relazione valga rispetto a **tutti** i nodi dei rispettivi sottoalberi.

Teorema 8.3 – Visita inorder e ordinamento. La visita inorder di un BST produce le chiavi in ordine non decrescente.

Dimostrazione. Per induzione sulla struttura dell'albero. Se l'albero è vuoto, la sequenza è vuota (ordinata per definizione). Altrimenti, per ipotesi induttiva, la visita inorder del sottoalbero sinistro produce le chiavi del sottoalbero sinistro in ordine. Per la proprietà BST, tutte queste chiavi sono $< x$.key. Analogamente, la visita inorder del sottoalbero destro produce chiavi $\geq x$.key in ordine. La concatenazione (sottoalbero sinistro, radice, sottoalbero destro) è dunque ordinata. ■

8.5.2 Operazioni fondamentali

Tutte le operazioni fondamentali su un BST hanno complessità $O(h)$, dove h è l'altezza dell'albero.

8.5.2.1 Ricerca

Definizione 8.9 – Ricerca in un BST. Data una chiave k e un BST T, l'operazione `Search(T, k)` restituisce un puntatore al nodo con chiave k, oppure NIL se la chiave non è presente.

Versione ricorsiva:

treeSearch

```
Node treeSearch(Node x, int k){
    if((x == NIL) || (k == x.key)){
        return x;
    }
    if(k < x.key){
        return treeSearch(x.left, k);
    } else {
        return treeSearch(x.right, k);
    }
}
```

Versione iterativa:

iterativeTreeSearch

```
Node iterativeTreeSearch(Node x, int k){
    while((x != NIL) && (k != x.key)){
        if(k < x.key){
            x := x.left;
        } else {
            x := x.right;
        }
    }
    return x;
}
```

> **Nota.** La versione iterativa è generalmente preferita perché evita il costo dell'overhead delle chiamate ricorsive ed è più efficiente in termini di spazio (non usa lo stack di ricorsione).

Complessità: $O(h)$, dove h è l'altezza dell'albero. Ad ogni passo scendiamo di un livello, quindi il numero massimo di confronti è $h + 1$.

8.5.2.2 Minimo e Massimo

> **Definizione 8.10 – Minimo e Massimo.** In un BST, il **minimo** è il nodo raggiunto seguendo sempre il figlio sinistro a partire dalla radice. Il **massimo** è il nodo raggiunto seguendo sempre il figlio destro.

treeMinimum / treeMaximum

```
Node treeMinimum(Node x){
    while(x.left != NIL){
        x := x.left;
    }
    return x;
}

Node treeMaximum(Node x){
    while(x.right != NIL){
        x := x.right;
    }
    return x;
}
```

Complessità: $O(h)$ per entrambe le operazioni.

> **Osservazione.** La correttezza segue direttamente dalla proprietà BST: tutte le chiavi nel sottoalbero sinistro sono strettamente minori della radice, e tutte le chiavi nel sottoalbero destro sono maggiori o uguali.

8.5.2.3 Successore e Predecessore

> **Definizione 8.11 – Successore.** Il **successore** di un nodo x in un BST è il nodo con la più piccola chiave maggiore di x.key, cioè il nodo che segue x nella visita inorder.

Si distinguono due casi:
1. Se x ha un figlio destro, il successore è il **minimo del sottoalbero destro**.
2. Se x non ha figlio destro, il successore è il **primo antenato** di x il cui figlio sinistro è anch'esso antenato di x (cioè risaliamo finché non siamo un figlio sinistro).

treeSuccessor

```
Node treeSuccessor(Node x){
    if(x.right != NIL){
        return treeMinimum(x.right);
    }
    Node y = x.parent;
    while((y != NIL) && (x == y.right)){
        x := y;
        y := y.parent;
    }
    return y;
}
```

Predecessore: simmetricamente, il predecessore è il massimo del sottoalbero sinistro (se esiste), altrimenti il primo antenato di cui x è nel sottoalbero destro.

treePredecessor

```
Node treePredecessor(Node x){
    if(x.left != NIL){
        return treeMaximum(x.left);
    }
    Node y = x.parent;
    while((y != NIL) && (x == y.left)){
        x := y;
        y := y.parent;
    }
    return y;
}
```

Complessità: $O(h)$ per entrambe le operazioni, poiché si percorre al più un cammino dalla radice a una foglia.

8.5.2.4 Inserimento

> **Definizione 8.12 – Inserimento in un BST.** L'operazione `Insert(T, z)` inserisce un nuovo nodo z nel BST T mantenendo la proprietà BST. Il nodo viene inserito come **foglia** nella posizione corretta.

treeInsert

```
treeInsert(Tree T, Node z){
    Node y = NIL;
    Node x = T.root;
    while(x != NIL){
        y := x;
        if(z.key < x.key){
            x := x.left;
        } else {
            x := x.right;
        }
    }
    z.parent := y;
    if(y == NIL){
        T.root := z;         // l'albero era vuoto
    } else {
        if(z.key < y.key){
            y.left := z;
        } else {
            y.right := z;
        }
    }
    z.left := NIL;
    z.right := NIL;
}
```

> **Nota.** La variabile y mantiene il puntatore al **padre** del nodo corrente x (tecnica detta «trailing pointer»). Quando x diventa NIL, y è il nodo a cui agganciare il nuovo nodo z. Si noti che le chiavi uguali vengono inserite nel sottoalbero destro, coerentemente con la proprietà BST adottata.

Complessità: $O(h)$, poiché scendiamo dalla radice fino a una posizione vuota.

8.5.2.5 Cancellazione

La cancellazione è l'operazione più complessa su un BST. Si distinguono tre casi in base alla struttura del nodo z da eliminare.

> **Definizione 8.13 – Cancellazione da un BST.** L'operazione `Delete(T, z)` rimuove il nodo z dal BST T mantenendo la proprietà BST.

Caso 1 – z non ha figli (foglia): si rimuove z semplicemente aggiornando il puntatore del padre.

Caso 2 – z ha un solo figlio: si elimina z e si collega il figlio direttamente al padre di z.

Caso 3 – z ha due figli: si sostituisce z con il suo **successore** y (il minimo del sottoalbero destro di z). Siccome y non ha figlio sinistro (altrimenti non sarebbe il minimo), si ricade nel Caso 1 o 2 per rimuovere y dalla sua posizione originale.

Per semplificare il codice, definiamo una procedura ausiliaria che sostituisce un sottoalbero con un altro:

transplant

```
transplant(Tree T, Node u, Node v){
    if(u.parent == NIL){
```

```
        T.root := v;
    } else {
        if(u == u.parent.left){
            u.parent.left := v;
        } else {
            u.parent.right := v;
        }
    }
    if(v != NIL){
        v.parent := u.parent;
    }
}
```

Utilizzando `transplant`, la cancellazione è:

treeDelete

```
treeDelete(Tree T, Node z){
    if(z.left == NIL){
        // Caso 1 o 2: nessun figlio sinistro
        transplant(T, z, z.right);
    } else {
        if(z.right == NIL){
            // Caso 2: solo figlio sinistro
            transplant(T, z, z.left);
        } else {
            // Caso 3: due figli
            Node y = treeMinimum(z.right);
            if(y.parent != z){
                transplant(T, y, y.right);
                y.right := z.right;
                y.right.parent := y;
            }
            transplant(T, z, y);
            y.left := z.left;
            y.left.parent := y;
        }
    }
}
```

Complessità: $O(h)$, poiché le operazioni `treeMinimum` e `transplant` richiedono ciascuna $O(h)$ nel caso peggiore.

8.5.3 Analisi della complessità

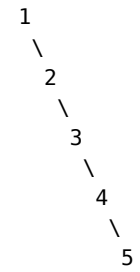
Le prestazioni di un BST dipendono criticamente dalla sua altezza h.

Operazione	Complessità	Descrizione
`Search(T, k)`	$O(h)$	Ricerca di una chiave
`Minimum(T)`	$O(h)$	Chiave minima
`Maximum(T)`	$O(h)$	Chiave massima
`Successor(x)`	$O(h)$	Successore di un nodo
`Predecessor(x)`	$O(h)$	Predecessore di un nodo
`Insert(T, z)`	$O(h)$	Inserimento di un nodo
`Delete(T, z)`	$O(h)$	Cancellazione di un nodo

Tabella 25: Complessità delle operazioni su BST in funzione dell'altezza h

8.5.3.1 Caso peggiore

Se le chiavi vengono inserite in ordine crescente (o decrescente), il BST degenera in una lista concatenata. In questo caso $h = n - 1$ e tutte le operazioni hanno complessità $O(n)$.

> **Esempio 8.7 – BST degenere.** Inserendo le chiavi $1, 2, 3, 4, 5$ in quest'ordine si ottiene:
>
> 
>
>
> L'altezza è $h = 4 = n - 1$, e la ricerca di 5 richiede 5 confronti.

8.5.3.2 Caso migliore

Se l'albero è **bilanciato** (i sottoalberi sinistro e destro di ogni nodo differiscono in altezza di al più 1), allora $h = \Theta(\log n)$ e tutte le operazioni hanno complessità $O(\log n)$.

8.5.3.3 Caso medio

> **Teorema 8.4 – Altezza media di un BST.** L'altezza attesa di un BST costruito inserendo n chiavi distinte in ordine casuale uniformemente distribuito è $O(\log n)$.

> **Nota.** La dimostrazione completa di questo risultato richiede strumenti probabilistici avanzati. L'idea chiave è che un inserimento casuale produce una struttura analoga a quella del QuickSort randomizzato: la radice partiziona le chiavi e la profondità attesa è logaritmica.

Riassumendo:

Caso	Altezza h	Complessità operazioni
Peggiore (degenere)	$n - 1$	$O(n)$
Migliore (bilanciato)	$\Theta(\log n)$	$O(\log n)$
Medio (casuale)	$O(\log n)$	$O(\log n)$

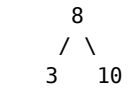
Tabella 26: Altezza e complessità nei diversi casi

8.5.4 Esempio completo

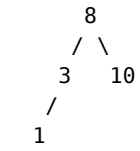
Mostriamo passo per passo la costruzione di un BST e le operazioni su di esso.

> **Esempio 8.8 – Costruzione di un BST.** Inseriamo le chiavi $8, 3, 10, 1, 6, 14, 4, 7, 13$ in quest'ordine.
>
> **Passo 1** – Inserimento di 8 (radice):
>
> 
>
>
> **Passo 2** – Inserimento di 3 ($3 < 8$, va a sinistra):
>
> 
>

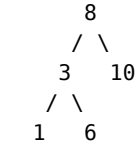
Passo 3 – Inserimento di 10 ($10 \geq 8$, va a destra):



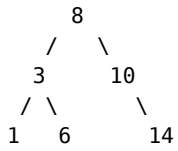
Passo 4 – Inserimento di 1 ($1 < 8$, $1 < 3$, va a sinistra di 3):



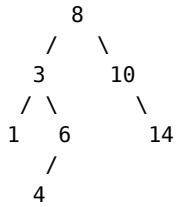
Passo 5 – Inserimento di 6 ($6 < 8$, $6 \geq 3$, va a destra di 3):



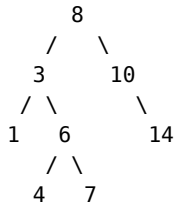
Passo 6 – Inserimento di 14 ($14 \geq 8$, $14 \geq 10$, va a destra di 10):



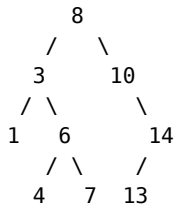
Passo 7 – Inserimento di 4 ($4 < 8$, $4 \geq 3$, $4 < 6$, va a sinistra di 6):



Passo 8 – Inserimento di 7 ($7 < 8$, $7 \geq 3$, $7 \geq 6$, va a destra di 6):



Passo 9 – Inserimento di 13 ($13 \geq 8$, $13 \geq 10$, $13 < 14$, va a sinistra di 14):



Esempio 8.9 – Operazioni sul BST. Sull'albero costruito nell'esempio precedente eseguiamo le seguenti operazioni.

Ricerca di 6: $8 \to 3 \to 6$ (trovato dopo 3 confronti).

Ricerca di 5: $8 \to 3 \to 6 \to 4 \to$ NIL (non trovato, 4 confronti).

Minimo: $8 \to 3 \to 1$ (il minimo è 1).

Massimo: $8 \to 10 \to 14$ (il massimo è 14).

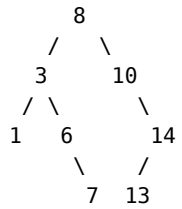
Successore di 6: il nodo 6 ha un figlio destro (7), quindi il successore è il minimo del sottoalbero destro, cioè 7.

Successore di 7: il nodo 7 non ha figlio destro, risaliamo: $7 \to 6$ (eravamo a destra), $6 \to 3$ (eravamo a destra), $3 \to 8$ (eravamo a sinistra, ci fermiamo). Il successore è 8.

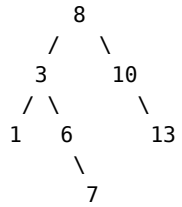
Predecessore di 10: il nodo 10 non ha figlio sinistro, risaliamo: $10 \to 8$ (eravamo a destra, ci fermiamo). Il predecessore è 8.

Esempio 8.10 – Cancellazione dal BST. Partiamo dall'albero completo e mostriamo i tre casi di cancellazione.

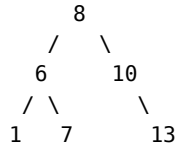
Caso 1 – Cancellazione di 4 (foglia): si rimuove il nodo e si imposta il figlio sinistro di 6 a NIL.



Caso 2 – Cancellazione di 14 (un solo figlio): si sostituisce 14 con il suo unico figlio 13.



Caso 3 – Cancellazione di 3 (due figli): il successore di 3 è 6 (minimo del sottoalbero destro). Si sostituisce 3 con 6:



La proprietà BST è mantenuta: $1 < 6 < 7 < 8 < 10 < 13$.

Nota. Per ottenere BST con altezza garantita $O(\log n)$ nel caso peggiore, si utilizzano alberi bilanciati come gli **AVL** o i **Red-Black Tree**. Queste strutture aggiungono operazioni di **rotazione** dopo inserimenti e cancellazioni per mantenere il bilanciamento, garantendo complessità $O(\log n)$ per tutte le operazioni.