

Programmazione ed Algoritmica

Appunti del corso di Laurea in Informatica

Diego Stefanini

Anno Accademico 2025-2026

Indice

1	Introduzione	3
2	Linguaggi Formali e Grammatiche	8
3	Semantica e Linguaggio MAO	15
3.1	Regole di Inferenza e Sistemi Logici	15
3.2	Induzione	17
3.3	Esercizi: Dimostrazioni Induttive su Grammatiche	18
4	Complessità Computazionale	52
5	Algoritmi di Ordinamento	60
6	Strutture Dati	87

1 Introduzione

1.0.1 Cos'è l'Informatica

Definizione 1 (Informatica). Lo studio sistematico dei processi **algoritmici** che descrivono e trasformano le **informazioni**: la loro teoria, analisi, progettazione, efficienza, implementazione e applicazione (ACM)

Nota (ACM). Association for Computing Machinery è l'associazione accademica internazionale di scienziati ed educatori dell'informatica.

1.0.2 Algoritmo

Definizione 2 (Algoritmo). Sequenza **finita** di **passi univocamente determinati** che se eseguiti da un **esecutore** portano alla risoluzione di un **problema**

passi \leadsto istruzioni, operazioni elementari

esecutore \leadsto calcolatore

Le tre componenti fondamentali:

1. **Problema** da risolvere
2. **Procedimento** da seguire
3. **Esecutore** che esegue le istruzioni

Nota. La descrizione dell'algoritmo deve essere comprensibile dall'esecutore

1.0.2.1 Proprietà degli algoritmi

- **Finitezza:** un algoritmo è costituito da un numero finito di passi e deve terminare in tempo finito
- **Non ambiguità:** i passi sono univocamente determinati, senza ambiguità o scelte arbitrarie
- **Determinismo:** ogni istruzione ha un solo passo successivo possibile, date le condizioni attuali
- **Generalità:** risolve una classe di problemi, non una singola istanza

Esempio 1 (Algoritmo: trovare il massimo). **Problema:** dato un insieme di n numeri, trovare il più grande.

Algoritmo (in linguaggio naturale):

1. Considera il primo numero come «massimo corrente»
2. Per ogni numero successivo:
 - Se è maggiore del massimo corrente, diventa il nuovo massimo
3. Alla fine, restituisci il massimo corrente

In MAO:

```
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;
}

```

1.0.3 Programma e programmazione

Definizione 3 (Programma). Formulazione di un algoritmo in un linguaggio di programmazione, indica al calcolatore quali operazioni eseguire, in quale ordine, con quali dati e sotto quali condizioni.

Definizione 4 (Programmare). Scrivere istruzioni che un computer può eseguire per risolvere problemi o svolgere compiti specifici.

1.0.3.1 Differenza tra algoritmo e programma

Aspetto	Algoritmo	Programma
Livello	Astratto	Concreto
Linguaggio	Naturale/pseudo-codice	Linguaggio di programmazione
Esecutore	Umano o macchina	Solo macchina
Dettagli	Omessi	Tutti specificati

Tabella 1: Confronto algoritmo vs programma

1.0.4 Computer

Definizione 5 (Computer). Macchine che eseguono semplici operazioni, **rapidamente** e con grande **precisione**.

Un computer riceve in **input** un programma (testo) e un insieme di dati e produce in **output** il risultato dell'esecuzione del programma. A differenza di altre macchine automatiche, essi sono **programmabili**: il compito dipende dal programma.

1.0.5 Problem solving

Definizione 6 (Problem Solving). Attività finalizzata all'analisi e alla risoluzione dei *problemi computazionali*

Le fasi del problem solving:

1. **Specifica**: definizione del problema (input/output)
2. **Progettazione**: ideazione dell'algoritmo
3. **Codifica**: traduzione in linguaggio di programmazione
4. **Testing**: verifica della correttezza
5. **Esecuzione**: run del programma

Nota. Ci sono problemi che non hanno algoritmi che li risolvano! (es. Problema della fermata)

1.0.6 Problemi computazionali

Definizione 7 (Problema computazionale). Un problema formulato matematicamente di cui cerchiamo una soluzione algoritmica. È definito da:

- **Input:** i dati in ingresso (istanza del problema)
- **Output:** il risultato atteso
- **Relazione:** vincolo tra input e output

Esempio 2 (Problema: ordinamento).

- **Input:** sequenza di n numeri $\langle a_1, a_2, \dots, a_n \rangle$
- **Output:** permutazione $\langle a'_1, a'_2, \dots, a'_n \rangle$ tale che $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Esempio 3 (Problema: ricerca).

- **Input:** sequenza di n numeri e un valore k
- **Output:** indice i tale che $A[i] = k$, oppure -1 se k non è presente

1.0.7 Linguaggi di Programmazione

1.0.7.1 Perché serve un linguaggio di programmazione

- I linguaggi di programmazione servono a tradurre idee in istruzioni eseguibili
- Sono formali, precisi e non ambigui
- Esistono molti linguaggi (Python, Java, C, JavaScript...), tutti condividono concetti di base

Nota. Un linguaggio di programmazione è un **ponte** tra il pensiero umano e l'esecuzione della macchina.

1.0.7.2 Il nostro linguaggio: MAO

Definizione 8 (MAO). Modello Astratto Operazionale: linguaggio di programmazione semplificato, leggibile e didattico.

MAO è simile a linguaggi reali come JavaScript o C, ma senza complicazioni tecniche. Serve a studiare i **concetti fondamentali** della programmazione.

Esempio 4 (Primo programma in MAO).

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

Questo programma dichiara due variabili intere e calcola la loro somma.

1.0.7.3 Sintassi vs Semantica

Lo studio di un linguaggio comprende due aspetti distinti:

Aspetto	Domanda	Esempio
Sintassi	Come si scrive?	<code>if(x > 0){ ... }</code>
Semantica	Cosa significa?	Se $x > 0$, esegui il blocco

Tabella 2: Sintassi vs Semantica

Definizione 9 (Sintassi). L'insieme di regole che definiscono la **struttura grammaticale** delle frasi valide del linguaggio.

Definizione 10 (Semantica). Il **significato** delle frasi sintatticamente corrette: cosa fa il programma quando viene eseguito.

Esempio 5 (Stesso significato, sintassi diversa). In MAO: `x := x + 1;`

In Python: `x = x + 1`

In C++: `x++;`

Tutte queste istruzioni hanno la stessa semantica: incrementano x di 1.

1.0.7.4 Obiettivi della programmazione

- **Correttezza:** il programma fa quello che deve fare
- **Efficienza:** usa poche risorse (tempo, memoria)
- **Leggibilità:** altri programmatori possono capirlo
- **Manutenibilità:** facile da modificare e aggiornare

1.0.7.5 Sintassi formale: perché?

Nel linguaggio naturale possiamo capire frasi con errori:

- «Dmoani vado al mrae» → «Domani vado al mare»

Un computer invece ha bisogno di **regole precise**. Non può «intuire» cosa intendevamo.

Esempio 6 (Errore sintattico).

```
int x = ;    // ERRORE: manca il valore
if x > 0    // ERRORE: mancano le parentesi
```

1.0.7.6 Come definire formalmente un linguaggio?

Un linguaggio può essere visto come un **insieme di frasi ben formate**. Per descrivere questo insieme si usano le **grammatiche formali**.

Nota. Le grammatiche formali non riguardano solo la programmazione: anche l'italiano ha una grammatica (soggetto + verbo + complemento).

1.0.7.7 Elementi base di MAO

Elemento	Esempio
Dichiarazione	<code>int x = 5;</code>
Assegnamento	<code>x := x + 1;</code>
Condizionale	<code>if(x > 0){ ... } else { ... }</code>
Ciclo	<code>while(x > 0){ ... }</code>
Funzione	<code>int f(int a){ return a * 2; }</code>

Tabella 3: Costrutti base di MAO

Nota (Attenzione). In MAO usiamo = per la **dichiarazione** e := per l'**assegnamento**. Questa distinzione è importante!

2 Linguaggi Formali e Grammatiche

2.0.1 Alfabeti, Stringhe e Linguaggi

2.0.2 Alfabeto

Definizione 1. Un alfabeto A è un insieme finito di simboli (chiamati terminali)

Esempio 1 (alfabeto dei caratteri dell'alfabeto italiano). $A_1 = \{a, b, c, d, \dots, z\}$

Esempio 2 (alfabeto delle cifre binarie). $A_2 = \{0, 1\}$

2.0.3 Stringa

Una stringa di un alfabeto A è una sequenza di lunghezza finita di simboli dell'alfabeto

Definizione 2. $\forall a \in A, a_1 a_2 \dots a_n$ con $n \geq 0$

2.0.3.1 Lunghezza di stringa

Il numero naturale n è detto lunghezza della stringa e si denota con $|s|$

Nota. Se $n = 0$ la stringa s è chiamata stringa vuota e viene rappresentata con ε

Esempio 3. $|abfbz| = 5$
 $|\varepsilon| = 0$

2.0.3.2 Stringhe di lunghezza fissata

Definizione 3. Definiamo l'insieme A^n come l'insieme di tutte e sole le stringhe sull'alfabeto A che hanno lunghezza n

Esempio 4. $A = \{0, 1\}$
 $A^0 = \{\varepsilon\}$
 $A^1 = A = \{0, 1\}$
 $A^2 = \{00, 01, 11, 10\}$

2.0.3.3 Stringhe sull'alfabeto

Definiamo l'insieme A^* come l'insieme di tutte le stringhe sull'alfabeto A

Definizione 4. $A^* = \bigcup_{n \geq 0} A^n = \{\varepsilon\} \cup A^1 \cup A^2 \cup A^3 \cup \dots$

Nota. Se A non è vuoto, A^* è infinito!

2.0.4 Un linguaggio

Un linguaggio L su un alfabeto A è un sottoinsieme di A^*

Definizione 5. $L \subseteq A^*$

Nota. Linguaggi particolari:

- linguaggio vuoto: \emptyset
- linguaggio di tutte le possibili stringhe su A : A^*

2.0.5 Descrizione dei linguaggi

Definizione 6. Descrivere un linguaggio di programmazione significa descrivere l'insieme delle stringhe ben formate del linguaggio, che chiamiamo programmi

Nota. I programmi ammissibili sono infiniti!

È possibile identificare l'insieme delle stringhe ben formate che caratterizzano un linguaggio come:

- Insieme delle stringhe **generate** da una grammatica: **metodo generativo**
- Insieme delle stringhe **riconosciute** da un automa: **metodo riconoscitivo**

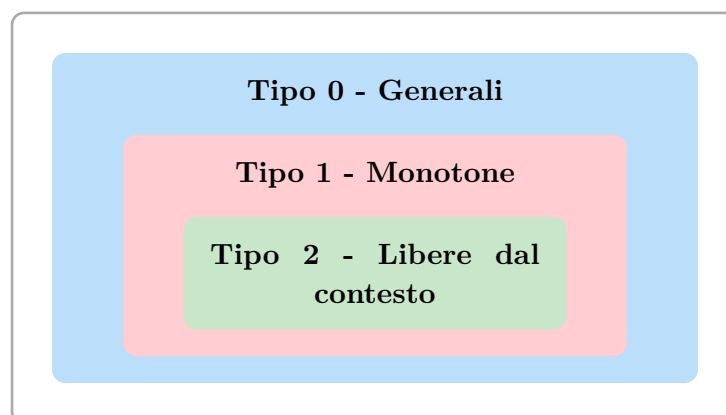
2.0.6 Le grammatiche formali

La grammatica è una tripla $G = (T, N, P)$ dove:

- T è l'insieme dei simboli **terminali** che sostituiscono l'alfabeto di riferimento
- N è l'insieme dei simboli **non terminali** che introducono le categorie sintattiche
- con $S = T \cup N$, $P \subseteq S^* \times S^*$, che è l'insieme delle **produzioni** della grammatica con forma $\alpha \Rightarrow \beta$, dove α contiene **almeno** un **non-terminale**

2.0.7 Gerarchia di Chomsky

Le grammatiche possono essere classificate in base alla forma delle loro produzioni.





2.0.8 Backus-Naur form (BNF)

È un modo compatto per scrivere una grammatica: tutte le produzioni che si riferiscono allo stesso non terminale sono raggruppate utilizzando il simbolo «|» come separatore.

Esempio 5. Direzione ::= sinistra | destra
 Consiglio ::= svolta a Direzione | prosegui dritto
 Percorso ::= Consiglio | Consiglio, poi Percorso

2.0.8.1 Linguaggio delle espressioni aritmetiche

con questo insieme $L \subseteq A^* = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \times, +\}^*$ possiamo descrivere il linguaggio delle espressioni aritmetiche con una grammatica.

Esempio 6 (N). Cifra ::= 0 | 1 | 2 | ... | 9
 Num ::= Cifra | Num Cifra
 Op ::= + | ×
 Esp ::= Num | Esp Op Esp

Nota. Ogni produzione definisce come costruire espressioni valide

2.0.9 Derivazioni e Linguaggi Generati

2.0.10 Derivativo immediato

Definizione 7. Data una grammatica $G = (T, N, P)$, e date due stringhe $\beta, \beta' \in S^*$ di simboli terminali e non terminali, si dice che β' è un **derivativo immediato** di β , scrivendo $\beta \rightarrow \beta'$ se e solo se partendo da β è possibile ottenere β' applicando una produzione:

- β contiene occorrenza di un qualche simbolo non terminale: $\beta = \beta_1 X \beta_2$ per stringhe $\beta_1, \beta_2 \in S^*$ e $X \in N$
- esiste una produzione $X ::= \alpha$ tale che $\beta_1 = \beta_1 \alpha \beta_2$

Esempio 7. $\beta = \text{Exp} + \text{Exp Op } 2$, se $\text{Exp} ::= \text{Num}$ allora $\beta' = \text{Exp} + \text{Num Op } 2$

2.0.11 Derivativo

Definizione 8. Data una grammatica $G = (T, N, P)$, e date due stringhe $\beta, \beta' \in S^*$ di simboli terminali e non terminali, si dice che β' è un **derivativo** di β , scrivendo $\beta \xrightarrow{*} \beta'$ se e solo se partendo da β è possibile ottenere β' applicando un numero qualsiasi di produzioni, passo dopo passo ($\beta \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow \dots \rightarrow \beta_n = \beta'$)

2.0.12 Linguaggio generato

Definizione 9. Data una grammatica $G = (T, N, P)$ e un simbolo non terminale $X \in N$, il linguaggio di X , scritto $L(X)$ è l'insieme di tutti e soli i derivativi $\beta \in T^*$ di X
 $L(X) = \{w \mid w \in T^*, X \xrightarrow{*} w\}$

2.0.12.1 Quali stringhe appartengono al linguaggio?

Data una grammatica $G = (T, N, P)$ una stringa di terminali $w \in T^*$ appartiene al linguaggio $L(X)$ del non terminale $X \in N$ se partendo dal simbolo X e applicando una o più volte le regole di produzione, si può ottenere la stringa $w \in T^*$ composta da solo terminali. Invece non appartiene al linguaggio se partendo dal simbolo X e indipendentemente da come si applicano le regole di produzione, non è possibile generare i terminali della stringa.

Esempio 8 (N). Riprendendo l'esempio dei numeri naturali

Cifra ::= 0 | 1 | 2 | ... | 9

Num ::= Cifra | Num Cifra

Op ::= + | ×

Esp ::= Num | Esp Op Esp

Modifichiamolo affinché non possa generare numeri che hanno 0 come cifra più significativa:

NonZero ::= 1 | 2 | 3 | ... | 9

Cifra ::= 0 | NonZero

Pos ::= Cifra | Pos Cifra

Num ::= 0 | Pos

2.0.13 Alberi di Derivazione

2.0.14 Derivazione Canonica

Definizione 10 (Derivazione canonica sinistra). Una derivazione è **canonica sinistra** se, ad ogni passo, il simbolo non terminale sostituito è sempre quello più a **sinistra** nella stringa.

Definizione 11 (Derivazione canonica destra). Una derivazione è **canonica destra** se, ad ogni passo, il simbolo non terminale sostituito è sempre quello più a **destra** nella stringa.

Esempio 9 (Derivazioni canoniche a confronto). Data la grammatica:

- Exp ::= Num | Exp + Exp
- Num ::= 0 | 1 | 2 | ... | 9

Deriviamo la stringa $3 + 5 + 2$:

Derivazione canonica sinistra (sostituiamo sempre il non-terminale più a sinistra):

$$\begin{aligned}
 & \text{Exp} \\
 & \rightarrow \text{Exp} + \text{Exp} \\
 & \rightarrow \text{Exp} + \text{Exp} + \text{Exp} \\
 & \rightarrow \text{Num} + \text{Exp} + \text{Exp} \\
 & \rightarrow 3 + \text{Exp} + \text{Exp} \\
 & \rightarrow 3 + \text{Num} + \text{Exp} \\
 & \rightarrow 3 + 5 + \text{Exp} \\
 & \rightarrow 3 + 5 + \text{Num} \\
 & \rightarrow 3 + 5 + 2
 \end{aligned}$$

Derivazione canonica destra (sostituiamo sempre il non-terminale più a destra):

$$\begin{aligned}
 & \text{Exp} \\
 & \rightarrow \text{Exp} + \text{Exp} \\
 & \rightarrow \text{Exp} + \text{Num} \\
 & \rightarrow \text{Exp} + 2 \\
 & \rightarrow \text{Exp} + \text{Exp} + 2 \\
 & \rightarrow \text{Exp} + \text{Num} + 2 \\
 & \rightarrow \text{Exp} + 5 + 2 \\
 & \rightarrow \text{Num} + 5 + 2 \\
 & \rightarrow 3 + 5 + 2
 \end{aligned}$$

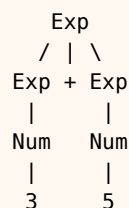
Nota. Entrambe le derivazioni producono la stessa stringa, ma con ordini di sostituzione diversi.

2.0.15 Alberi di derivazione

Definizione 12 (Albero di derivazione (Parse Tree)). Un **albero di derivazione** è una rappresentazione grafica della derivazione di una stringa che astrae dall'ordine di applicazione delle produzioni:

- La **radice** è il simbolo iniziale
- I **nodi interni** sono non-terminali
- Le **foglie** sono terminali
- I figli di un nodo corrispondono alla parte destra di una produzione

Esempio 10 (Costruzione dell'albero di derivazione). Per la stringa $3 + 5$ con la grammatica $\text{Exp} ::= \text{Num} \mid \text{Exp} + \text{Exp}$:



L'albero si legge così:

- Exp si espande in $\text{Exp} + \text{Exp}$

- Il primo Exp diventa Num, che diventa 3
- Il secondo Exp diventa Num, che diventa 5

Nota. 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.

2.0.16 Valutazione degli alberi

La valutazione di un'espressione segue la struttura dell'albero:

- Si valutano prima i **sottoalberi** (ricorsivamente)
- Poi si applica l'operatore alla radice

Esempio 11 (Valutazione). Per l'albero di $3 + 5$:

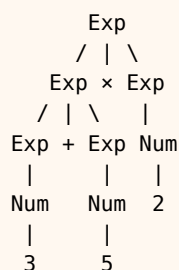
1. Valuta il sottoalbero sinistro: 3
2. Valuta il sottoalbero destro: 5
3. Applica l'operatore $+$: $3 + 5 = 8$

2.0.17 Ambiguità

Definizione 13 (Grammatica ambigua). Una grammatica è **ambigua** se esiste almeno una stringa del linguaggio che ammette **due o più alberi di derivazione** distinti.

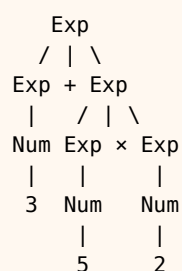
Esempio 12 (Ambiguità: $3 + 5 \times 2$). Con la grammatica $\text{Exp} ::= \text{Num} \mid \text{Exp} + \text{Exp} \mid \text{Exp} \times \text{Exp}$, la stringa $3 + 5 \times 2$ ha due alberi possibili:

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



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

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



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

Nota (Conseguenza dell'ambiguità). La stessa stringa può avere **significati diversi!** Questo è un problema grave per un linguaggio di programmazione.

2.0.18 Risoluzione dell'ambiguità

Per eliminare l'ambiguità si può:

1. **Modificare la grammatica** introducendo livelli di precedenza
2. **Usare parentesi** per forzare l'ordine di valutazione

Esempio 13 (Grammatica non ambigua con precedenza). $\text{Exp} ::= \text{Term} \mid \text{Exp} + \text{Term}$

$\text{Term} ::= \text{Factor} \mid \text{Term} \times \text{Factor}$

$\text{Factor} ::= \text{Num} \mid (\text{Exp})$

$\text{Num} ::= 0 \mid 1 \mid \dots \mid 9$

Con questa grammatica, $3 + 5 \times 2$ ha un solo albero possibile che rispetta la precedenza: \times prima di $+$.

Nota. Non sempre è possibile eliminare l'ambiguità: esistono **linguaggi inerentemente ambigui** per cui ogni grammatica che li genera è ambigua.

3 Semantica e Linguaggio MAO

3.1 Regole di Inferenza e Sistemi Logici

Data una produzione $S ::= a \mid b \mid a S b$ possiamo scriverla in due modi:

- lettura generativa (o produttiva): la grammatica è vista come un insieme di regole di riscrittura, la produzione si legge da sinistra verso destra e se trovo il simbolo S posso rimpiazzarlo con $a S b$
- lettura induttiva (costruttiva): la grammatica è una definizione induttiva, la produzione si legge da destra verso sinistra e ogni produzione è una clausola che spiega come costruire stringhe valide; presa una stringa w qualsiasi in $L(S)$ concludo che anche $a w b$ è in $L(S)$

3.1.1 Regole di inferenza

Per definire la semantica del linguaggio Mao verranno utilizzate le regole di inferenza. Date premesse p e conclusione q possiamo definire le regole di inferenza come:

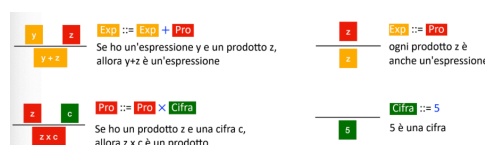
$$\frac{p_1 \dots p_n}{q} \quad (\text{Nome-Regola})$$

Se tutte le premesse sono vere allora si può trarre la conclusione q ; se le premesse sono vuote allora si ha un assioma di forma (assioma).

$$\frac{}{q} \quad (\text{Nome-assioma})$$

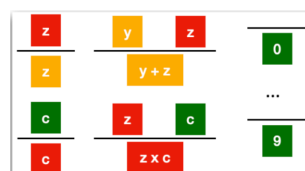
Le regole di inferenza vengono utilizzate per derivare nuovi giudizi a partire dalle verità già note.

3.1.1.1 Produzioni come regole di inferenza

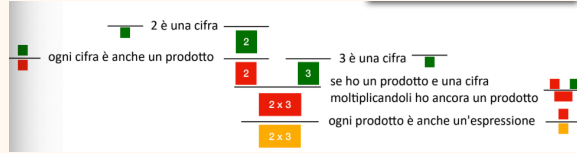


3.1.2 Sistemi logici

Un sistema logico è un insieme di regole di inferenza che possono essere applicate per dimostrare la validità di formule.



Esempio 1. Prendendo come riferimento la grammatica nell'immagine precedente, 2×3 è un'espressione valida?

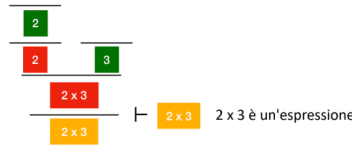


3.1.2.1 Derivazione

Definizione 1. Una derivazione nel sistema logico è una sequenza di passaggi che partendo dagli assiomi e applicando le regole giustificano una certa conclusione, scritto $d \vdash q$, si legge «la derivazione d dimostra il giudizio q» oppure «d è una derivazione per q»

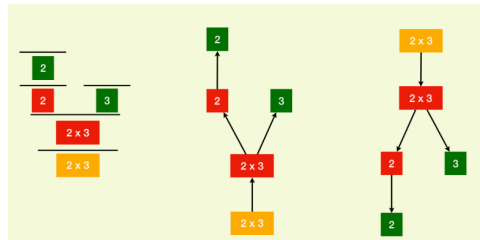
Le derivazioni sono definite come segue:

- ogni assioma del sistema logico è una derivazione per q
- se $\frac{p_1 \dots p_n}{q}$ è regola del sistema logico le cui premesse sono derivabili $d_1 \vdash p_1, \dots, d_n \vdash p_n$, allora $\frac{d_1 \dots d_n}{q}$



3.1.2.2 Alberi di derivazioni

Anche le derivazioni formano una struttura ad albero, anche se in questo caso la radice è posta verso il basso.



3.1.2.3 Grammatiche come sistemi logici

Formalmente invece dei blocchetti colorati, introduciamo le formule $x \in L(X)$, col significato che la stringa x appartiene al linguaggio di X.

$$\frac{y \quad z}{y+z} \quad \frac{y \in L(\text{Exp}) \quad z \in L(\text{Prod})}{y+z \in L(\text{Exp})}$$

Ad ogni produzione della grammatica (dove $\omega_i \in T^*$ e $X_i \in N$) associamo una regola di inferenza

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

3.1.2.4 Valutazione di espressioni

I sistemi logici permettono di assegnare un significato ad espressioni e comandi seguendone la struttura grammaticale.

Esempio 2. riprendendo la grammatica ambigua delle espressioni con sole cifre:

$$E ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid E + E \mid E \times E$$

presa un'espressione w : se w è una cifra allora il valore è il numero che rappresenta; se $w = w_1 + w_2$ è la somma di Espressione e il valore è la somma dei valori; analogamente vale anche per il prodotto.

Definiamo un predicato di valutazione per ogni categoria sintattica $w \Downarrow v$, spesso abbreviato con $w \Downarrow v$, si legge dicendo che «il termine $w \in T^*$ del linguaggio di $X \in N$ ha valore v

$$\begin{array}{c} \overline{0 \Downarrow 0} \quad \overline{1 \Downarrow 1} \quad \overline{2 \Downarrow 2} \quad \dots \quad \overline{9 \Downarrow 9} \\[10pt] \frac{x_1 \Downarrow n_1 \quad x_2 \Downarrow n_2}{x_1 + x_2 \Downarrow n} \quad n = n_1 + n_2 \quad \frac{x_1 \Downarrow n_1 \quad x_2 \Downarrow n_2}{x_1 \times x_2 \Downarrow n} \quad n = n_1 \times n_2 \\[10pt] \begin{array}{c} n_1 = 1 \quad \overline{1 \Downarrow n_1} \quad \overline{3 \times 2 \Downarrow n_2} \\ \hline 1 + (3 \times 2) \Downarrow n \end{array} \quad n = n_1 + n_2 \\[10pt] \begin{array}{c} \overline{1 \Downarrow 1} \quad \overline{3 \Downarrow 3} \quad \overline{2 \Downarrow 2} \\ \hline \overline{1 \Downarrow 1} \quad \overline{3 \times 2 \Downarrow 6} \quad \overline{2 \Downarrow 2} \\ \hline 1 + (3 \times 2) \Downarrow 7 \end{array} \quad \begin{array}{c} 6 = 3 \times 2 \\ 7 = 1 + 6 \end{array} \end{array}$$

3.2 Induzione

Definizione 2. 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 ci permette di costruire un numero infinito di un insieme mediante un numero finito di regole, definire il comportamento di una funzione su un insieme infinito di elementi descrivendo il comportamento su casi finiti e dimostrare che una proprietà è valida per tutti gli elementi di un insieme finito, esaminando un numero finito di casi.

3.2.1 Induzione matematica

Voglio dimostrare che una proprietà è valida per tutti i naturali. Preso un generico n , assumendo che la proprietà sia vera per n , dimostro che è vera anche per $n + 1$.

Esempio 3. Voglio dimostrare che per ogni naturale positivo n , la somma dei primi n positivi è la metà del prodotto n e $n + 1$.

$$1 + 2 + 3 + 4 + \dots + n = \frac{n(n+1)}{2}$$

Caso induttivo: prendo un generico n , assumo valga $1 + 2 + 3 + 4 + \dots + n = \frac{n(n+1)}{2}$ e cerco di dimostrare che $1 + 2 + 3 + 4 + \dots + n + (n + 1) = \frac{(n+1)(n+2)}{2}$

parto da sinistra $1 + 2 + 3 + 4 + \dots + n + (n + 1) =$

applico l'ipotesi induttiva: $1 + 2 + 3 + 4 + \dots + n = \frac{n(n+1)}{2}$

metto a comun denominatore $\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$

porto $(n+1)$ in evidenza!

3.2.2 Induzione strutturale

Voglio dimostrare che una proprietà è valida per tutte le stringhe $s \in T^*$ di un linguaggio generato da una grammatica. Nel caso base dimostro la proprietà per le stringhe $\omega \in T^*$ che compaiono nella parte destra delle produzioni atomiche ($X ::= \omega$). Nei casi induttivi, presa una qualsiasi altra produzione non atomica dimostro che se la proprietà è valida per le stringhe $s_1, \dots, s_n \in T^*$, allora deve valere anche per $\omega_0 s_1 \omega_1 s_2 \dots s_n \omega_n$.

3.2.3 Induzione sulle derivazioni

Voglio dimostrare che una proprietà è valida per tutti i giudizi derivabili in un sistema logico. Nel caso base dimostro la proprietà è valida per tutte le conclusioni degli assiomi. Nel caso induttivo, presa una qualsiasi altra regola di inferenza dimostro che se la proprietà è valida per tutte le premesse allora vale anche per la conclusione.

Caso base: ovvio per tutti gli assiomi! $0 \Downarrow 0$ $2 \Downarrow 2$ $4 \Downarrow 4$ $6 \Downarrow 6$ $8 \Downarrow 8$

Caso induttivo: $\frac{x_1 \Downarrow n_1 \quad x_2 \Downarrow n_2}{x_1 + x_2 \Downarrow n} \quad n = n_1 + n_2$ assumendo che n_1 e n_2 siano pari, la loro somma n è pari

$\frac{x_1 \Downarrow n_1 \quad x_2 \Downarrow n_2}{x_1 \times x_2 \Downarrow n} \quad n = n_1 \times n_2$ assumendo che n_1 e n_2 siano pari, il loro prodotto n è pari

3.3 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.

3.3.1 Esercizio 1: Bilanciamento di a e b

Esempio 4 (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: $S \rightarrow ab$

La stringa generata è $w = ab$. Contiamo le occorrenze:

- $\#_a(ab) = 1$
- $\#_b(ab) = 1$

Quindi $\#_a(w) = \#_b(w) = 1$. ✓

Caso induttivo: $S \rightarrow aSb$

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:

- $\#_a(aw'b) = 1 + \#_a(w') = 1 + k$
- $\#_b(aw'b) = \#_b(w') + 1 = k + 1$

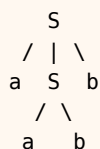
Quindi $\#_a(w) = \#_b(w) = k + 1$. ✓ ■

Esempio 5 (Alberi di derivazione). Mostriamo gli alberi di derivazione per alcune stringhe del linguaggio:

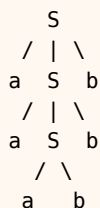
Stringa ab (caso base):



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



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



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 $L(S)$).

3.3.2 Esercizio 2: Stringhe con esattamente una c

Esempio 6 (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: $S \rightarrow c$

La stringa generata è $w = c$. Verifichiamo:

- $\#_c(c) = 1 \checkmark$
- $\#_a(c) = 0 = \#_b(c) \checkmark$

Caso induttivo 1: $S \rightarrow bSa$

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:

- $\#_c(bw'a) = \#_c(w') = 1 \checkmark$
- $\#_a(bw'a) = \#_a(w') + 1 = k + 1$
- $\#_b(bw'a) = 1 + \#_b(w') = 1 + k = k + 1$

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

Caso induttivo 2: $S \rightarrow aSb$

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:

- $\#_c(aw'b) = \#_c(w') = 1 \checkmark$
- $\#_a(aw'b) = 1 + \#_a(w') = 1 + k$
- $\#_b(aw'b) = \#_b(w') + 1 = k + 1$

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

Esempio 7 (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: $S \rightarrow aSb \rightarrow abSab \rightarrow abcab$

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

3.3.3 Esercizio 3: Almeno una *a*

Esempio 8 (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: $S \rightarrow a$

La stringa generata e' $w = a$. Verifichiamo:

- $\#_a(a) = 1 \geq 1 \checkmark$

Caso induttivo 1: $S \rightarrow Sa$

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

La nuova stringa e' $w = w'a$. Verifichiamo:

- $\#_a(w'a) = \#_a(w') + 1 \geq 1 + 1 = 2 \geq 1 \checkmark$

Caso induttivo 2: $S \rightarrow Sb$

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

La nuova stringa e' $w = w'b$. Verifichiamo:

- $\#_a(w'b) = \#_a(w') \geq 1 \checkmark$

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

Esempio 9 (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

```

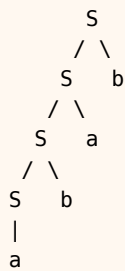
Stringa *aa*:

```

  S
 / \
S   a
|
a

```

Stringa *abab*:



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

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 .

3.3.4 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 $X ::= \alpha_1 Y_1 \alpha_2 Y_2 \dots Y_n \alpha_{n+1}$ 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 $P(\alpha_1 w_1 \alpha_2 w_2 \dots w_n \alpha_{n+1})$ vale

3.3.5 MiniMao

3.3.6 Ambiente e memoria, dichiarazioni, assegnamenti e composizione sequenziale

La programmazione consiste nell'ideare uno o più algoritmi che risolvano un problema (problem solving), 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 un risultato di un dato. Per riferire questi valori si utilizzano nomi simbolici detti nomi di variabile.

Nota. Equazioni matematiche e assegnamenti sono due concetti diversi. Prendendo in esempio $n = n^2 - 2$, se considerato un'equazione il calcolo da eseguire è cercare il valore di n che verifica l'uguaglianza $n \in \{2, -1\}$. Invece se esso viene considerato un assegnamento, se per esempio n vale 5, dopo il calcolo n vale 23.

3.3.7 Concetto di variabile

Una variabile è come una scatola con nome e tipo.

```
int età;
int età = 15;
```

Nel tempo può cambiare valore:

```
età := 16;
```

3.3.8 Stato del programma

Possiamo avere molte variabili nel nostro programma, ognuna con il suo valore. Lo stato è l'insieme di tutte le variabili e i loro valori.

Esempio 10.

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

3.3.9 Sintassi di MiniMao

Introduzione del linguaggio Mao, in versione ristretta. Per prima cosa verrà definita la sintassi usando grammatiche formali, in un secondo momento verrà definita la semantica utilizzando sistemi logici.

Le categorie sintattiche di MiniMao sono:

- Valori (V): dati elementari come numeri interi o booleani
- Identificatori (Id): simboli per riferirsi a variabili, procedure ecc
- Espressione (E): combinano V, Id ed operatori per produrre altri valori
- Tipi (T): indicano l'insieme dei valori ammissibili
- Comandi (C): descrivono le azioni da eseguire

Definizione 3 (Valori V). Inizialmente consideriamo solo programmi che manipolano valori interi (\mathbb{Z}) e booleani (\mathbb{B})

Esempio 11. $V ::= n \mid \text{true} \mid \text{false}$

Definizione 4 (Identificatori Id). Nomi simbolici che permettono al programmatore di riferirsi in modo chiaro e univoco a valori, procedure, funzioni ecc. In Mao sono stringhe alfanumeriche e possono contenere l'underscore (`_`)

Esempio 12. `mio_id1`

Nota. Alcune parole chiave sono riservate e non sono identificatori validi

Definizione 5 (Espressioni E). Sono combinazioni di valori, identificatori e operatori che producono un nuovo valore booleano o intero.

$E ::= V \mid \text{Id} \mid E \text{ bop } E \mid \text{uop } E \mid (E)$

Definizione 6 (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}
×	moltiplicazione	$\mathbb{Z} \times \mathbb{Z}$	\mathbb{Z}
÷	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}
≤	minore o uguale	$\mathbb{Z} \times \mathbb{Z}$	\mathbb{B}
>	maggiore	$\mathbb{Z} \times \mathbb{Z}$	\mathbb{B}
≥	maggiore o uguale	$\mathbb{Z} \times \mathbb{Z}$	\mathbb{B}
==	uguaglianza	$\mathbb{Z} \times \mathbb{Z} \text{ o } \mathbb{B} \times \mathbb{B}$	\mathbb{B}
≠	disuguaglianza	$\mathbb{Z} \times \mathbb{Z} \text{ o } \mathbb{B} \times \mathbb{B}$	\mathbb{B}
∧	and logico	$\mathbb{B} \times \mathbb{B}$	\mathbb{B}
∨	or logico	$\mathbb{B} \times \mathbb{B}$	\mathbb{B}

Tabella 4: Operatori binari in MiniMao

Nota. La divisione intera ÷ e il modulo mod richiedono che il secondo operando sia diverso da zero.

Definizione 7 (Operatori unari). Gli operatori unari prendono un solo operando.

Simbolo	Nome	Tipo operando	Tipo risultato
−	negazione aritmetica	\mathbb{Z}	\mathbb{Z}
¬	negazione logica	\mathbb{B}	\mathbb{B}

Tabella 5: Operatori unari in MiniMao

Definizione 8 (Tipi T). Può essere tipo base o tipo composto. Per ogni tipo assumiamo un valore di default (0 per interi e false per bool).

$$T ::= T_b \mid T_c$$

Definizione 9 (Comandi C). $C ::= \text{skip} \mid T \text{ Id} = E \mid \text{Id} := E \mid CC \mid \{C\} \mid \text{if}(E)\{C\} \text{ else } \{C\} \mid \text{while}(E)\{C\}$

Esempio 13 (sintassi di MiniMao).

```
int x = 1;
int n = 0;
```

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

3.3.10 Semantica di MiniMao

La semantica è uno strumento che ci permette di ragionare formalmente sul comportamento di un programma e quindi di studiare proprietà di programmi come correttezza, equivalenza, terminazione, divergenza ecc...

Definizione 10 (Semantica operativa). La semantica operativa descrive il comportamento di un programma in termini dei passi che compie per eseguirlo un'opportuna macchina astratta

Definizione 11 (Macchina astratta). Una macchina astratta è una macchina semplificata ideale, il cui stato è dato dagli oggetti ambiente ρ e la memoria σ

3.3.11 Ambiente e memoria

L'ambiente $\rho : \mathbb{I} \hookrightarrow \mathbb{L}$ è una funzione parziale da identificatori a locazioni.

La memoria $\sigma : \mathbb{L} \hookrightarrow \mathbb{V}$ è una funzione parziale da locazioni a valori.

Esempio 14. $\rho(\text{eta}) = 0, \rho(\text{studente}) = 1 \rightarrow \sigma(0) = 15, \sigma(1) = \text{true}$

Nota. Una funzione parziale significa che potrebbe non essere definita su tutti gli argomenti.

3.3.12 Stato

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

Se le funzioni parziali sono definite su un numero finito di argomenti, le possiamo rappresentare elencando tutte le possibili associazioni argomento-valore $\rho = [x \mapsto l_1, y \mapsto l_2], \sigma = [l_1 \mapsto 15, l_2 \mapsto \text{true}]$

3.3.13 Predicati semantici

Per definire la semantica dei programmi abbiamo bisogno di definire come valutare espressioni ed eseguire comandi. A tal fine introduciamo:

- $\langle E, \rho, \sigma \rangle \Downarrow v$ che si legge «l'espressione E nello stato (ρ, σ) ha valore v »
- $\langle C, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$ che si legge «l'esecuzione del comando C nello stato iniziale (ρ, σ) termina producendo lo stato finale (ρ', σ') »

3.3.14 Espressioni pure o con effetti collaterali

Le espressioni con effetti collaterali sono espressioni la cui valutazione può comportare modifiche di memoria, come allocazione o modifica di celle già presenti. La valutazione in questo caso si esprime come $\langle E, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

Le espressioni pure non alterano lo stato. MiniMao ammette solo espressioni pure.

3.3.15 Semantica operativa (prima parte)

Le regole semantiche definiscono formalmente come si valutano espressioni e si eseguono comandi.

3.3.15.1 Regole per le espressioni

$$\begin{array}{c}
 \frac{}{\langle n, \rho, \sigma \rangle \Downarrow n} \quad (\text{Val-Int}) \\
 \\
 \frac{}{\langle \text{true}, \rho, \sigma \rangle \Downarrow \text{true}} \quad (\text{Val-True}) \\
 \\
 \frac{}{\langle \text{false}, \rho, \sigma \rangle \Downarrow \text{false}} \quad (\text{Val-False}) \\
 \\
 \frac{\rho(\text{Id}) = l \quad \sigma(l) = v}{\langle \text{Id}, \rho, \sigma \rangle \Downarrow v} \quad (\text{Val-Var}) \\
 \\
 \frac{\langle E_1, \rho, \sigma \rangle \Downarrow v_1 \quad \langle E_2, \rho, \sigma \rangle \Downarrow v_2 \quad v = v_1 \text{ bop } v_2}{\langle E_1 \text{ bop } E_2, \rho, \sigma \rangle \Downarrow v} \quad (\text{Val-Bop}) \\
 \\
 \frac{\langle E, \rho, \sigma \rangle \Downarrow v' \quad v = \text{uop } v'}{\langle \text{uop } E, \rho, \sigma \rangle \Downarrow v} \quad (\text{Val-Uop})
 \end{array}$$

Per eseguire una dichiarazione nello stato (ρ, σ) si deve:

- valutare il valore v di E nello stato.
- preparare una cella di memoria $l \notin \sigma$
- estendere la memoria σ con l'associazione $l \mapsto v$
- estendere l'ambiente ρ con l'associazione $\text{Id} \mapsto l$
- per essere sicuri controllare che il valore v sia di tipo T

Per eseguire un assegnamento nello stato (ρ, σ) dobbiamo:

- valutare v di E nello stato
- individuare la cella di memoria l che l'ambiente ρ associa a Id
- aggiornare la memoria σ con l'associazione $l \mapsto v$
- essere sicuri che v sia di tipo T di Id

Per eseguire una sequenza di comandi nello stato:

- eseguire C_1 nello stato (ρ, σ) ottenendo (ρ_1, σ_1)
- eseguire C_2 nello stato (ρ_1, σ_1) ottenendo (ρ_2, σ_2) , chiamato stato finale di $C_1 C_2$

3.3.16 Semantica operativa dei comandi (base)

3.3.16.1 Skip

$$\frac{}{\langle \text{skip};, \rho, \sigma \rangle \rightarrow \langle \rho, \sigma \rangle} \quad (\text{Cmd-Skip})$$

3.3.16.2 Dichiarazione

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

3.3.16.3 Assegnamento

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

3.3.16.4 Sequenza

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

3.3.17 Sviluppo sequenziale

Lo sviluppo sequenziale ci permette di eseguire i calcoli in uno stile più leggibile, rispetto alle derivazioni.

Esempio 15 (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):

$$\frac{\frac{\langle x, \rho, \sigma \rangle \Downarrow 5}{\langle y := x; \rho, \sigma \rangle \rightarrow \langle \rho, \sigma[l_2 \mapsto 5] \rangle} \quad \frac{\langle y, \rho, \sigma' \rangle \Downarrow 5}{\langle x := y; \rho, \sigma' \rangle \rightarrow \langle \rho, \sigma'[l_1 \mapsto 5] \rangle}}{\langle y:=x; x:=y; \rho, \sigma \rangle \rightarrow \langle \rho, \sigma'' \rangle}$$

Come sviluppo sequenziale (lineare):

$$\begin{aligned} &\{x \mapsto 5, y \mapsto 3\} \\ &\quad \downarrow y := x; \\ &\{x \mapsto 5, y \mapsto 5\} \\ &\quad \downarrow x := y; \\ &\{x \mapsto 5, y \mapsto 5\} \end{aligned}$$

In sviluppo sequenziale descriviamo l'esecuzione come sequenza di stati:

$$\begin{aligned} &\langle C, \rho_0, \sigma_0 \rangle \\ &\quad \downarrow \\ &\langle \rho_1, \sigma_1 \rangle \text{ (dopo primo comando)} \\ &\quad \downarrow \\ &\langle \rho_2, \sigma_2 \rangle \text{ (dopo secondo comando)} \\ &\quad \dots \\ &\quad \downarrow \\ &\langle \rho_n, \sigma_n \rangle \text{ (stato finale)} \end{aligned}$$

Notazione semplificata: invece di scrivere la configurazione completa, scriviamo solo lo stato come insieme di associazioni variabile \mapsto valore:

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

Questa notazione «fonde» ambiente e memoria: $x \mapsto v$ significa che la variabile x ha valore v .

Esempio 16 (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;`

- 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;`

- 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;`

- 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;`

- 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 $y = 4$.

Esempio 17 (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: $\rho = [a \mapsto l_1, b \mapsto l_2, \text{max} \mapsto l_3], \sigma = [l_1 \mapsto 5, l_2 \mapsto 3, l_3 \mapsto 0]$

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

Poiché la guardia è vera, eseguiamo il ramo `then`:

Esecuzione `max := a;`

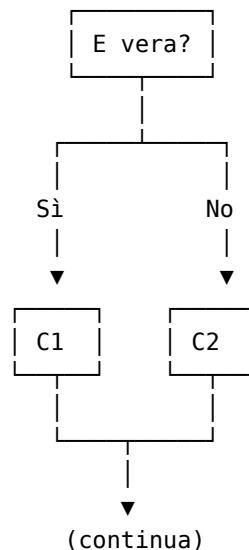
- Valuto a : $\sigma(\rho(a)) = \sigma(l_1) = 5$
- Aggiorno: $\sigma' = \sigma[l_3 \mapsto 5] = [l_1 \mapsto 5, l_2 \mapsto 3, l_3 \mapsto 5]$

Stato finale: `max = 5` ✓

3.3.18 Blocchi e comandi condizionali

3.3.19 Comando condizionale

I comandi condizionali servono per prendere decisioni all'interno dei programmi.



3.3.20 Comando condizionale in Mao

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

E è espressione booleana, se vera si esegue C_1 , altrimenti si esegue C_2 .

Esempio 18 (comando condizionale in Mao).

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

3.3.21 Comando if-then

Sostanzialmente è un comando if-else ma dove eseguiamo un comando solamente se la guardia (E) è vera.

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

Esempio 19 (comando condizionale if-then).

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

Nota. Le parentesi tonde e graffe svolgono due compiti distinti all'interno del linguaggio. Le parentesi tonde vengono utilizzate per fissare un ordine di valutazione.

3.3.22 Il blocco {C}

Tutte le variabili dichiarate all'interno del blocco sono visibili solo al suo interno. Lo scope 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 del blocco è visibile in tutti i comandi successivi nello stesso blocco e all'interno dei blocchi annidati.

3.3.23 Shadowing

È possibile dichiarare una variabile all'interno di un blocco con il nome di una variabile dichiarata in precedenza. In questo caso la variabile dichiarata all'interno del blocco nasconde quella dichiarata all'esterno.

3.3.24 Semantica operativa (seconda parte)

3.3.24.1 Blocco

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

Nota. Il blocco «dimentica» le variabili dichiarate al suo interno: l'ambiente finale è quello iniziale ρ , ma la memoria σ' mantiene le modifiche.

3.3.24.2 Condizionale (if-then-else)

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

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

3.3.25 Cicli

3.3.25.1 Comando iterativo (while)

Il ciclo while ripete l'esecuzione del corpo finché la guardia è vera.

3.3.26 Comando iterativo in Mao

`while (E) {C}`

E è un'espressione booleana, se vera viene eseguito C e si controlla la condizione, se falsa termina.

Esempio 20 (comando iterativo in Mao).

```
while( cioccolatini > 0) {
    cioccolatini := cioccolatini - 1
}
```

3.3.27 Semantica del while

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

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

Nota. La regola (Cmd-WhileTrue) è **ricorsiva**: dopo aver eseguito il corpo, si esegue di nuovo il while sullo stato risultante.

3.3.28 Riepilogo regole semantiche

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 6: Riepilogo delle regole semantiche di MiniMao

Esempio 21 (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:

- Guardia: $\langle n > 0, \rho, \sigma_0 \rangle \Downarrow 3 > 0 = \text{true} \checkmark$
- $s := s + n$: $s = 0 + 3 = 3 \rightarrow \sigma_1 = [l_1 \mapsto 3, l_2 \mapsto 3]$
- $n := n - 1$: $n = 3 - 1 = 2 \rightarrow \sigma_{1'} = [l_1 \mapsto 2, l_2 \mapsto 3]$

Iterazione 2:

- Guardia: $2 > 0 = \text{true} \checkmark$
- $s := s + n$: $s = 3 + 2 = 5 \rightarrow \sigma_2 = [l_1 \mapsto 2, l_2 \mapsto 5]$
- $n := n - 1$: $n = 2 - 1 = 1 \rightarrow \sigma_{2'} = [l_1 \mapsto 1, l_2 \mapsto 5]$

Iterazione 3:

- Guardia: $1 > 0 = \text{true} \checkmark$
- $s := s + n$: $s = 5 + 1 = 6 \rightarrow \sigma_3 = [l_1 \mapsto 1, l_2 \mapsto 6]$
- $n := n - 1$: $n = 1 - 1 = 0 \rightarrow \sigma_{3'} = [l_1 \mapsto 0, l_2 \mapsto 6]$

Iterazione 4:

- Guardia: $0 > 0 = \text{false} \times \rightarrow$ ciclo termina

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

Esempio 22 (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: $\langle n > 0, \rho, \sigma_0 \rangle \Downarrow 4 > 0 = \text{true} \checkmark$
- $f := f * n$: $\langle f * n, \rho, \sigma_0 \rangle \Downarrow 1 \times 4 = 4$
 - $\sigma_{0'} = \sigma_0[l_f \mapsto 4] = [l_n \mapsto 4, l_f \mapsto 4]$
- $n := n - 1$: $\langle n - 1, \rho, \sigma_{0'} \rangle \Downarrow 4 - 1 = 3$
 - $\sigma_1 = \sigma_{0'}[l_n \mapsto 3] = [l_n \mapsto 3, l_f \mapsto 4]$

Iterazione 2:

- Guardia: $\langle n > 0, \rho, \sigma_1 \rangle \Downarrow 3 > 0 = \text{true} \checkmark$
- $f := f * n$: $\langle f * n, \rho, \sigma_1 \rangle \Downarrow 4 \times 3 = 12$
 - $\sigma_{1'} = \sigma_1[l_f \mapsto 12] = [l_n \mapsto 3, l_f \mapsto 12]$
- $n := n - 1$: $\langle n - 1, \rho, \sigma_{1'} \rangle \Downarrow 3 - 1 = 2$
 - $\sigma_2 = \sigma_{1'}[l_n \mapsto 2] = [l_n \mapsto 2, l_f \mapsto 12]$

Iterazione 3:

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

Iterazione 4:

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

Iterazione 5:

- Guardia: $\langle n > 0, \rho, \sigma_4 \rangle \Downarrow 0 > 0 = \text{false} \times \rightarrow \text{ciclo termina}$

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 7: Evoluzione dello stato nel calcolo di $4!$

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

Esempio 23 (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 e: $\rho_1 = [x \mapsto l_1]$
- La memoria corrente e: $\sigma_1 = [l_1 \mapsto 10]$

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 perche 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

- Applichiamo la 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 e piu accessibile.

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 <code>int x = 10;</code>	$[x \mapsto l_1]$	l_1 (valore 10)
Dentro blocco, dopo <code>int x = 5;</code>	$[x \mapsto l_2]$	l_2 (valore 5)
Dentro blocco, dopo <code>x := x + 1;</code>	$[x \mapsto l_2]$	l_2 (valore 6)
Dopo uscita dal blocco	$[x \mapsto l_1]$	l_1 (valore 10)
Dopo <code>x := x * 2;</code>	$[x \mapsto l_1]$	l_1 (valore 20)

Tabella 8: Evoluzione dell'ambiente con shadowing

Esempio 24 (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
 - $\rho_4 = [a \mapsto l_a, b \mapsto l_b, c \mapsto l_c, \text{max} \mapsto l_m]$
 - $\sigma_4 = [l_a \mapsto 7, l_b \mapsto 3, l_c \mapsto 5, l_m \mapsto 7]$

Stato dopo le dichiarazioni:

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

Passo 5: Primo condizionale `if (b > max) { max := b; }`

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

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

Passo 6: Secondo condizionale `if (c > max) { max := c; }`

- Valuto la guardia: $\langle c > \text{max}, \rho_4, \sigma_5 \rangle$
 - $\sigma_5(\rho_4(c)) = \sigma_5(l_c) = 5$
 - $\sigma_5(\rho_4(\text{max})) = \sigma_5(l_m) = 7$
 - $5 > 7 = \text{false}$
- Poiche la guardia e falsa, applichiamo (Cmd-IfFalse)
- Lo stato **non cambia**: $\sigma_6 = \sigma_5$

Stato finale:

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

Il risultato e corretto: $\text{max} = 7 = \text{max}(7, 3, 5) \checkmark$

—

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

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

Primo condizionale `if (b > max) { max := b; }:`

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

Secondo condizionale `if (c > max) { max := c; }:`

- Guardia: $5 > 8 = \text{false}$
- Stato invariato: $\{a \mapsto 2, b \mapsto 8, c \mapsto 5, \text{max} \mapsto 8\}$

Risultato: $\text{max} = 8 = \text{max}(2, 8, 5) \checkmark$

—

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

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

Primo condizionale:

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

Secondo condizionale:

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

Risultato: $\text{max} = 9 = \text{max}(2, 3, 9) \checkmark$

3.3.29 Terminazione vs Divergenza

Con l'introduzione dei cicli adesso i programmi possono divergere, senza che essi producano un risultato finale.

Definizione 12 (Divergenza). La divergenza si riferisce alla situazione in cui un programma o un ciclo non terminano mai, continuando a eseguire istruzioni indefinitamente

Definizione 13 (Terminazione). La terminazione è una proprietà desiderabile nei programmi, infatti essa garantisce che il programma completi la sua esecuzione e produca un risultato in tempo finito. La terminazione è possibile solo se la guardia di un'iterazione contiene almeno una variabile e il corpo contenga almeno un assegnamento su quella variabile.

Nota. Non esiste un algoritmo che per ogni programma sia in grado di decidere se termina o no

3.3.30 Costrutti iterativi alternativi

In Mao si vede solamente il costrutto iterativo `while` ma ci sono diversi costrutti iterativi che i linguaggi di programmazione comuni mettono a disposizione. Ognuno di questi costrutti possono essere espressi usando un ciclo `while`.

3.3.31 Ciclo for

Costrutto particolarmente compatto, in una riga è possibile leggere quante iterazioni eseguirà.

Esempio 25 (for in JavaScript).

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

3.3.32 Ciclo do-while

Si utilizza per eseguire il corpo una volta senza controllare la condizione.

Esempio 26 (do-while in JavaScript).

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

3.3.33 Mao (Modello Astratto Operazionale)

3.3.34 Array e espressioni con effetti collaterali

Definizione 14 (array - informale). Un array è una struttura dati volta a trattare un insieme di dati omogenei in modo efficiente

Utilizzando `int[]` o `bool[]` e stabilito che una dichiarazione avviene come

$$\text{int}[] \text{voti} = [x_1, x_2, x_3]$$

Viene riservato un primo elemento in l_b (indirizzo base) per memorizzare la lunghezza dell'array. Per ciascun valore della serie lo si accede attraverso la sua posizione x :

$$l_b[x]$$

Nella struttura memoria-ambiente, si rappresenta nel seguente modo l'indirizzo base e le successive:

$$\begin{aligned}\rho(x) &= l_x, \sigma(l_x) = l_b \\ \sigma(l_b) &= 3 \\ \sigma(l_b[0]) &= 18\end{aligned}$$

Definizione 15 (Array - formale). Un array è una collezione finita dello stesso tipo, memorizzati contigualmente. Il numero degli elementi è la lunghezza dell'array. Tipo e lunghezza sono fissati alla dichiarazione e sono statici (non possono essere cambiati una volta dichiarati)

Un array permette di trattare come entità atomiche intere collezioni e al contempo permette di accedere ai singoli elementi tramite indici posizionali con intervallo

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

3.3.35 Sintassi

Per ottenere la lunghezza di un array è possibile utilizzare il costrutto `.length`

Esempio 27 (lunghezza array).

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

voti.length = 3
```

La sintassi degli array in Mao è così grammaticalmente rappresentata:

$$\begin{aligned} T &::= \dots \mid T[] \\ E &::= \dots \mid [S] \mid \text{new } T [E] \mid E.\text{length} \mid E [E] \\ S &::= E \mid E, S \\ C &::= \dots \mid E [E] := E \end{aligned}$$

Dove:

- $T[]$: tipo array di elementi di tipo T
- $[S]$: letterale array (serie di espressioni)
- $\text{new } T [E]$: allocazione di un nuovo array
- $E.\text{length}$: lunghezza dell'array
- $E [E]$: accesso a un elemento

3.3.36 Valori e riferimenti

Se in MiniMao le celle di memoria contenevano solamente valori interi e booleani

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

adesso in Mao si memorizzano anche gli indirizzi base degli array

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

Ora si includono tra i valori anche le locazioni (riferimenti)

3.3.37 Espressioni «pure» e non

Se in MiniMao la presenza nella grammatica di sole espressioni pure ci permetteva la seguente semplificazione

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

Adesso con l'introduzione di allocazione di memoria non è più possibile. Le espressioni che allocano memoria (come `new T[E]` o letterali array `[E1, E2, ...]`) producono anche una memoria modificata:

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

3.3.38 Semantica degli Array

3.3.38.1 Allocazione di un array letterale

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

Nota. L'array viene memorizzato con la lunghezza in l_b e gli elementi in $l_b + 1, \dots, l_b + n$.

3.3.38.2 Allocazione con new

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

3.3.38.3 Lunghezza

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

3.3.38.4 Accesso

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

3.3.38.5 Assegnamento in array

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

3.3.39 Aliasing

Esempio 28 (Problema dell'aliasing). Con l'aliasing si intende la copia di un array per riferimento:

```
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:

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

Quindi $a[0]$ e $b[0]$ accedono alla stessa cella di memoria.

Nota (Conseguenza). Quando si passa un array come parametro a una funzione, si passa il **riferimento**. Modifiche all'array nel corpo della funzione si riflettono sull'array originale.

3.3.40 Analisi statica e controllo di tipi

Grazie alle grammatiche le categorie sintattiche delle espressioni e dei comandi sono definite in maniera rigorosa, ma molti di essi, anche se sintatticamente validi, potrebbero portare ad un'altra categoria di eventi.

Definizione 16 (analisi statica). Con analisi statica si indicano i controlli fatti sul codice sorgente senza eseguire il programma

La maggior parte dei linguaggi moderni hanno diversi controlli con analisi statica, Mao si limita al controllo dei tipi (type checking) che permette di rilevare problemi legati alla mancanza di coerenza tra i tipi delle variabili, costanti e operazioni.

3.3.41 Sistemi di tipi

In Mao il controllo dei tipi viene in modo formale attraverso regole di tipo che stabiliscono le condizioni necessarie affinché espressione o comandi vengano considerati ben tipati. Il controllo

avviene in modo composizionale cioè le regole di tipo per espressioni e comandi sono definite su giudizi di tipo sulle sue componenti; per fare ciò è necessario sapere quali tipi sono associati alle variabili nelle espressioni o nel comando.

3.3.42 Ambiente di tipo

Definizione 17. L'ambiente di tipo $\Gamma : \mathbb{I} \hookrightarrow \mathbb{T}$ è una funzione parziale che assegna agli identificatori: $\Gamma(x) = \text{int}$ che significa assumere che x abbia tipo int

Scriviamo $\text{Id} : T$ per dire $\Gamma(\text{Id}) = T$.

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

Esempio 29. Dopo le seguenti dichiarazioni

```
int temp = 2;
bool y = true;
```

Si troverà $\Gamma = \{x : \text{int}, y : \text{bool}\}$

3.3.43 Variabili libere in un'espressione

Data un'espressione $E \in \mathbb{E}$, la funzione $\text{fv}(\cdot) : \mathbb{E} \rightarrow \mathcal{P}_{\text{fin}}(\mathbb{I})$ restituisce l'insieme di tutte le variabili che occorrono in E , dette variabili libere. La funzione viene definita per induzione nel modo seguente:

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

Esempio 30. $\text{fv}(x+3) = \text{fv}(x) \cup \text{fv}(3) = \{x\}$

Esempio 31. $\text{fv}(\text{new int}[a.\text{length}]) = \{a\}$

Esempio 32. $\text{fv}(x\%7 == z*x) = \{x, z\}$

Dato un comando $C \in \mathbb{C}$, la funzione $\text{fv}(\cdot) : \mathbb{C} \rightarrow \mathcal{P}_{\text{fin}}(\mathbb{I})$ restituisce l'insieme di tutte le variabili che occorrono in C senza essere dichiarate. La funzione è definita per induzione ma richiede $\text{dv}(\cdot) : \mathbb{C} \rightarrow \mathcal{P}_{\text{fin}}(\mathbb{I})$ che restituisce le variabili introdotte da dichiarazioni.

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

Dove $\text{dv}(C)$ è l'insieme delle variabili dichiarate in C :

$$\begin{aligned}
\text{dv}(T \text{ Id} = E;) &= \{\text{Id}\} \\
\text{dv}(C_1 C_2) &= \text{dv}(C_1) \cup \text{dv}(C_2) \\
\text{dv}(\text{altri comandi}) &= \emptyset
\end{aligned}$$

3.3.44 Giudizi di tipo per espressioni

Dato un ambiente di tipo Γ e una espressione E tale che $\text{fv}(E) \subseteq \text{dom}(\Gamma)$ possiamo derivare un giudizio di tipo, scritto $\Gamma \vdash E : T$ che si legge «nell'ambiente Γ , l'espressione E è ben tipata e ha tipo T »

3.3.45 Giudizi di tipo per i comandi

Dato un ambiente di tipo Γ e un comando C tale che $\text{fv}(C) \subseteq \text{dom}(\Gamma)$ possiamo derivare un giudizio di tipo, scritto $\Gamma \vdash C : \Gamma'$ che si legge «nell'ambiente Γ , il comando C è ben tipato e rende disponibile l'ambiente locale Γ' »

3.3.46 Regole di Type Checking per espressioni

3.3.46.1 Valori

$$\begin{array}{c}
\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \text{int}} \quad (\text{T-Int}) \\
\\
\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad (\text{T-True}) \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \quad (\text{T-False})
\end{array}$$

3.3.46.2 Variabili

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

3.3.46.3 Operatori aritmetici

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

3.3.46.4 Operatori di confronto

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

3.3.46.5 Operatori logici

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

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

3.3.47 Regole di Type Checking per comandi

3.3.47.1 Skip

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

3.3.47.2 Dichiarazione

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

3.3.47.3 Assegnamento

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

3.3.47.4 Sequenza

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

3.3.47.5 Blocco

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

3.3.47.6 Condizionale

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

3.3.47.7 While

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

Esempio 33 (Derivazione di tipo completa per espressione). Verifichiamo che l'espressione $x + y * 2$ sia ben tipata nell'ambiente $\Gamma = \{x : \text{int}, y : \text{int}\}$

Costruiamo l'albero di derivazione dal basso verso l'alto:

$$\frac{\frac{\frac{}{y : \text{int} \in \Gamma} \quad \frac{}{2 \in \mathbb{Z}}}{\Gamma \vdash y : \text{int} \quad \Gamma \vdash 2 : \text{int}}}{\Gamma \vdash y * 2 : \text{int}} \quad (\text{T-Mult})$$

E poi:

$$\frac{x : \text{int} \in \Gamma \quad \Gamma \vdash y * 2 : \text{int}}{\Gamma \vdash x + y * 2 : \text{int}} \quad (\text{T-Add})$$

L'espressione è **ben tipata** con tipo `int` ✓

Esempio 34 (Derivazione di tipo per comando). Verifichiamo che il comando sia ben tipato:

```
int z = 0;
z := x + 1;
```

Con $\Gamma_0 = \{x : \text{int}\}$

Passo 1: Tipo di `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: $\Gamma_1 = \Gamma_0 \cup \{z : \text{int}\} = \{x : \text{int}, z : \text{int}\}$

Passo 2: Tipo di `z := x + 1;`

Prima verifichiamo $x + 1$:

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

Poi l'assegnamento:

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

Il comando è **ben tipato** ✓

Esempio 35 (Errore di tipo). Consideriamo:

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

Con $\Gamma = \{x : \text{int}, y : \text{bool}\}$

Tentiamo di derivare il tipo di $x + y$:

- $\Gamma \vdash x : \text{int}$ ✓
- $\Gamma \vdash y : \text{bool}$ ✓
- Ma la regola (T-Add) richiede che **entrambi** gli operandi siano `int`!

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

Poiché y ha tipo `bool` \neq `int`, la derivazione **fallisce**.

Il programma è **mal tipato** e il compilatore segnala errore ✗

3.3.48 Esempi completi di derivazioni di type checking

Esempio 36 (Type checking di un programma con while). Verifichiamo il type checking del seguente programma:

```
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;`

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

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

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

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

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

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

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

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

Prima verifichiamo l'espressione `x + 2`:

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

Poi l'assegnamento:

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

E il blocco:

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

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

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

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

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

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

Passo 8: Verifica del condizionale completo:

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

Passo 9: Verifica del while completo:

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

Passo 10: Verifica della sequenza completa usando (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 è **ben tipato**.

Esempio 37 (Type checking della funzione abs). Verifichiamo che la funzione `abs` sia ben tipata:

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

Vogliamo dimostrare che $\Gamma \vdash \text{abs} : (\text{int}) \rightarrow \text{int}$

Passo 1: Costruiamo l'ambiente per il corpo della funzione.

Secondo la regola (T-Fun), dobbiamo verificare il corpo nell'ambiente:

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

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

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

Aggiorniamo l'ambiente: $\Gamma'' = \Gamma' \cup \{m : \text{int}\} = \Gamma \cup \{n : \text{int}, m : \text{int}\}$

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

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

Passo 4: Type checking dell'espressione `-m` in Γ''

L'operatore unario meno (negazione) richiede un operando intero:

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

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

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

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

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

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

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

Passo 8: Type checking di $\text{return } m$; in Γ''

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

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

Passo 9: Type checking della sequenza del corpo della funzione

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

Passo 10: Applicazione della regola (T-Fun)

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

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

Esempio 38 (Type checking di funzione ricorsiva su array). Verifichiamo il type checking della funzione ricorsiva:

```
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;
}
```

Questa funzione cerca l'elemento k nell'array a a partire dalla posizione p , e se lo trova lo azzerava.

Passo 1: Poiché la funzione è ricorsiva, usiamo la regola (T-RecFun).

L'ambiente per il corpo deve includere il tipo della funzione stessa:

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

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

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

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

Passo 3: Type checking della condizione $(p \geq 0) \ \&\& \ (p < a.\text{length})$ in Γ''

Prima verifichiamo $p \geq 0$:

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

Poi verifichiamo $a.\text{length}$:

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

Poi $p < a.\text{length}$:

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

Infine la congiunzione:

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

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 $\text{res} := \text{true}$;

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

Passo 8: Type checking della chiamata ricorsiva $\text{azzera}(a, k, p+1)$

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''):

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

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.

Passo 11: Applicazione della regola (T-RecFun)

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

dove C rappresenta il corpo della funzione.

Quindi $\Gamma \vdash \text{azzera} : (\text{int}[], \text{int}, \text{int}) \rightarrow \text{bool}$

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

3.3.49 Controllo di tipi vs inferenza di tipo

Il sistema di tipi di Mao è progettato per controllare che i tipi dichiarati dal programmatore per le variabili corrispondano all'uso che viene fatto di esse. Molti linguaggi di programmazione non richiedono di dichiarare il tipo di variabile al momento della sua dichiarazione:

- o perché non vengono controllati (JS)
- o perché viene utilizzato un meccanismo di inferenza di tipo (Golang): il contesto in cui viene usata la variabile determina il tipo

3.3.50 Tipi Base aggiuntivi - Char e stringhe

3.3.51 Caratteri

I caratteri sono utilizzati per rappresentare simboli, lettere e altri caratteri alfanumerici. In Mao il tipo `char` possono essere codificati ASCII o Unicode. I caratteri speciali vengono rappresentati da *sequenze di escape*.

Esempio 39.

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

3.3.52 Stringhe

Le stringhe in Mao sono un array di caratteri, di tipo `char[]`.

Esempio 40. ['C', 'i', 'a', 'o'] = "Ciao"

3.3.53 Assegnamento multiplo

Molti linguaggi permettono l'assegnamento multiplo, permettendo quindi di inizializzare più variabili contemporaneamente.

Esempio 41. `let x,y,z = 6,7,42;`

È anche possibile fare lo scambio delle variabili sfruttando l'assegnamento multiplo

`x, y := y,x;`

3.3.54 Sintassi assegnamento multiplo

$\text{LHS} ::= \text{Id} \mid \text{Id}, \text{LHS}$

$\text{RHS} ::= E \mid E, \text{RHS}$

Sfruttando queste nuove categorie sintattiche, generalizziamo i comandi atomici:

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

Dove LHS (Left-Hand Side) è la lista degli identificatori e RHS (Right-Hand Side) è la lista delle espressioni.

3.3.54.1 Type checking per 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 richiede che ogni espressione abbia il tipo corrispondente all'identificatore:

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

3.3.54.2 Semantica operativa assegnamento multiplo

Dichiarazione multipla:

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

Assegnamento multiplo:

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

Nota. L'assegnamento multiplo `x, y := y, x` permette lo swap in un solo comando perché tutte le espressioni RHS vengono valutate **prima** di effettuare gli assegnamenti.

3.3.55 Direttiva return

Permette di restituire il valore aggiornato di un'espressione qualsiasi.

Esempio 42.

`{ count := count + 1; return count; }`

La sintassi della direttiva `return` è la seguente:

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

Data la seguente definizione di variabili libere:

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

Per controllare il tipo dell'espressione:

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

La semantica operativa del `return`:

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

3.3.56 Funzioni

Le funzioni servono per dare un nome ad un frammento di codice che calcola un valore in modo da poter riutilizzare il codice tutte le volte che vogliamo utilizzando parametri diversi.

- **Definizione di funzione:** momento in cui scriviamo il codice dell'espressione che vogliamo calcolare. specifichiamo l'input parametrico (parametri formali)
- **Invocazione di funzione:** momento nel programma in cui chiamo la funzione per eseguire il calcolo dell'espressione specificata, i parametri sono effettivi (parametri attuali)

Esempio 43.

```
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);
}
```

3.3.57 Corrispondenza tra parametri formali e attuali

La chiamata di funzione deve contenere espressioni corrispondenti in numero e in tipo a quelle dichiarate nell'intestazione. La comunicazione avviene tramite il passaggio di parametri. Ad ogni parametro formale della funzione deve corrispondere un'espressione nella stessa posizione tra i parametri attuali.

3.3.57.1 Passaggio parametri per valore

Il passaggio per valore è la modalità più comune: ogni parametro formale viene inizializzato con una copia del valore corrispondente del parametro attuale. Se i parametri formali vengono modificati all'interno della funzione, tali modifiche non influenzano i parametri attuali originali.

3.3.57.2 Passaggio parametri per riferimento

Il passaggio per riferimento avviene inizializzando i parametri formali con il riferimento ai corrispondenti argomenti attuali. Se i parametri formali vengono modificati all'interno della funzione, tale modifica influenza i parametri attuali originali.

Nota. Quando un array viene passato come argomento attuale, stiamo implicitamente passando il riferimento all'array stesso l_b

3.3.58 Sintassi delle funzioni

La sintassi della dichiarazione di funzione:

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

Dove T_R è il tipo di ritorno (può essere `void` per funzioni senza valore di ritorno).

La sintassi della chiamata di funzione:

$$E ::= \dots \mid \text{Id}(E_1, \dots, E_n)$$

Il tipo di una funzione è rappresentato come:

$$(T_1, \dots, T_n) \rightarrow T_R$$

Dove (T_1, \dots, T_n) sono i tipi dei parametri formali e T_R è il tipo di ritorno.

Definizione variabili libere di una funzione:

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

Le variabili libere nel corpo della funzione escludono i parametri formali.

Regola di tipo per la dichiarazione di funzione:

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

Regola di tipo per l'invocazione:

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

3.3.59 Semantica operativa delle funzioni

3.3.59.1 Dichiarazione di funzione

La dichiarazione di una funzione memorizza la «chiusura» (closure) nell'ambiente:

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

Nota. La chiusura contiene: i nomi dei parametri, il corpo della funzione, e l'ambiente al momento della dichiarazione (per le variabili libere).

3.3.59.2 Chiamata di funzione

La chiamata di funzione:

1. Valuta gli argomenti
2. Crea un nuovo ambiente con i parametri formali legati ai valori degli argomenti
3. Esegue il corpo della funzione
4. Restituisce il valore del return

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

Nota. L'ambiente della funzione (ρ_f) viene esteso con i parametri, non l'ambiente del chiamante. Questo implementa lo **scoping statico**.

3.3.60 Ricorsione

In molti linguaggi di programmazione è ammessa la possibilità di definire funzioni ricorsive (funzione che richiama se stessa). La chiamata ricorsiva può essere:

- diretta: chiamata avviene direttamente in una espressione del corpo di f
- indiretta: f contiene una chiamata a g che a sua volta contiene una chiamata a f

Per avere possibile la ricorsione deve essere modificata la definizione di variabili libere:

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

Il nome della funzione stessa viene escluso dalle variabili libere, permettendo così la chiamata ricorsiva.

Regola di tipo per funzioni ricorsive (include l'associazione tra il nome della funzione e il suo tipo nell'ambiente):

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

Nota. La differenza rispetto a (T-Fun) è che nell'ambiente del corpo della funzione è presente anche il binding $\text{Id} : (T_1, \dots, T_n) \rightarrow T_R$, permettendo chiamate ricorsive ben tipate.

4 Complessità Computazionale

4.0.1 Complessità in tempo

Definizione 1 (Algoritmo). Sequenza finita di operazioni elementari (passi) univocamente determinati (non ambiguo, né operazione né la sequenza) che, se eseguita su un calcolatore, porta alla risoluzione di un problema.

Definizione 2 (Modello "RAM"). hanno costo unitario (costante):

- operazioni aritmetiche: $+$, $-$, \times , \div , $\%$
- operazioni di confronto: $<$, $>$, $==$, \neq , \dots
- operazioni logiche: \wedge , \vee , \neg , \dots
- operazioni di trasferimento: load, store, read, assegnamento
- operazioni di controllo: Return, chiamata di funzione

4.0.2 Esempi

4.0.3 Minimo in vettore

Input: array $A[1..n]$ di interi

Output: $i : 1 \leq i \leq n$ con $A[i]$ che è il valore più piccolo di A .

```
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;
}
```

Nel codice tutte le operazioni eseguite rientrano nei tipi di tempo di esecuzione costante. L'unica cosa che determina il tempo di esecuzione è il ciclo for, che è determinato dalla lunghezza dell'input. Il costo in tempo viene così definito: costante + $(n-1) * \text{costante}$ + costante. Si dice allora che la complessità è asintotica in tempo in funzione della dimensione dell'input ($n = |A|$)

4.0.4 Cerca-K

Input: $A[1..n]$ di interi, k intero.

Output: $\min i : A[i] = k$ oppure -1 se $k \notin A$

```
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;
    }
}

```

La complessità in tempo dipende da dove si trova k all'interno dell'input. Nel migliore dei casi è costante ($A[1] = k$), nel peggiore dei casi invece è lineare in n ($\nexists k$)

4.0.5 Costo computazionale di un algoritmo A

- **Tempo** $T_A(n)$: numero di operazioni elementari (passi) che esegue A , dipende dall'istanza di I :
 - Caso pessimo
 - Caso ottimo
- **Spazio**: numero di celle di memoria utilizzate durante l'esecuzione di A (anche quelle occupate dall'input)
- in funzione della **dimensione dell'input**

Ci concentreremo sulla complessità in tempo cercando di minimizzarla e, a parità di costo in tempo, cerchiamo di ridurre anche lo spazio.

Ci interessiamo al caso pessimo perché è un limite superiore al costo.

Ci interessiamo all'ordine di grandezza della funzione $T_A(n)$

Esempio 1.

$$T(n) = 3n + 2$$

costante moltiplicativa + termine di ordine inferiore

↓

$T(n)$ ha ordine di grandezza lineare

Esempio 2.

$$T(n) = 8n^2 + \log n + 4$$

termine quadratico + termini di ordine inferiore

↓

$T(n)$ ha complessità quadratica

4.0.6 Insertion Sort

L'insertion sort è un algoritmo di ordinamento con risoluzione con metodo iterativo.

Input: Array A di n numeri interi.

Output: A ordinato: $A[1] \leq A[2] \leq \dots \leq A[n]$

```

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;
}
}

```

4.0.7 Dimostrazione di correttezza con invariante di ciclo

L'invariante di ciclo è una proprietà conservata ad ogni iterazione di algoritmo. Ad ogni iterazione j^{esima} del for:

- il sottoarray $A[1..j-1]$ è ordinato
- il sottoarray (ordinato) $A[1..j-1]$ contiene gli stessi elementi che stavano in $A[1..j-1]$ iniziale.

Con queste due affermazioni possiamo dire che l'insertion sort è corretto.

4.0.8 Complessità

$T(n) = (n-1)(C_1 \times \text{numero esecuzioni while con guardia vera} + \text{while con guardia falsa} + C_5)$

- **Caso ottimo:** $d_j = 1 \ \forall j = 2..n \Rightarrow c(n) = \sum_{j=2}^n 1 = n-1$
Array in input già ordinato, il corpo del while non viene mai eseguito, quindi $c(n) = n-1 \Rightarrow \#$ confronti lineari.
- **Caso pessimo:** $d_j = j-1 \ \forall j = 2..n \Rightarrow c(n) = \sum_{j=2}^n j-1 = \sum_{j'=1}^{n-1} j' = \frac{n(n-1)}{2}$
Array in input ordinato in ordine inverso. $C(n) = \frac{n(n-1)}{2} = \binom{n}{2} \Rightarrow \#$ confronti quadratico in n

Per riassumere:

- lineare in caso ottimo
- quadratico in caso pessimo

4.0.9 Selection Sort

Il selection sort è un altro algoritmo di ordinamento iterativo al passo $i^{\text{esimo}} \ \forall i = 1..n-1$.

```

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;
    }
}

```

4.0.10 Complessità

$$T(n) = (n-1)(c_1 + (n-1)c_2)$$

confronti $= c(n) = \sum_{i=1}^{n-1} \times \sum_{j=i+1}^n 1 = \sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \dots + \frac{n(n-1)}{2} =$
complessità quadratica.

4.0.11 Invariante di ciclo

Definizione 3 (Invariante di ciclo). Un **invariante di ciclo** è una proprietà che:

1. È vera **prima** della prima iterazione (**inizializzazione**)
2. Se è vera prima di un'iterazione, rimane vera dopo (**mantenimento**)
3. Quando il ciclo termina, l'invariante fornisce una proprietà utile (**terminazione**)

4.0.11.1 Dimostrazione completa: Selection Sort

Invariante: All'inizio della i -esima iterazione:

- I primi $i-1$ elementi di A sono ordinati
- Questi $i-1$ elementi sono i più piccoli dell'array originale

Dimostrazione. Inizializzazione ($i = 1$):

- I «primi 0 elementi» sono banalmente ordinati (insieme vuoto)
- L'invariante è verificato

Mantenimento: Supponiamo l'invariante vero per i . Dimostriamo che vale per $i+1$.

- Il ciclo interno trova il minimo tra $A[i], A[i+1], \dots, A[n]$
- Questo minimo viene scambiato con $A[i]$
- Ora $A[1..i]$ contiene gli i elementi più piccoli, ordinati
- L'invariante vale per $i+1$

Terminazione ($i = n$):

- I primi $n-1$ elementi sono i più piccoli e sono ordinati
- L'elemento in $A[n]$ è quindi il più grande
- L'intero array è ordinato \square

■

4.0.11.2 Dimostrazione completa: Insertion Sort

Invariante: All'inizio dell'iterazione j -esima, il sottoarray $A[1..j-1]$ contiene gli elementi originali di $A[1..j-1]$, in ordine crescente.

Dimostrazione. Inizializzazione ($j = 2$):

- $A[1..1]$ contiene un solo elemento
- Un singolo elemento è banalmente ordinato \checkmark

Mantenimento: Supponiamo vero per j . Il ciclo interno:

- Prende $A[j]$ come chiave
- Sposta a destra gli elementi di $A[1..j-1]$ maggiori della chiave
- Inserisce la chiave nella posizione corretta
- Ora $A[1..j]$ è ordinato, quindi l'invariante vale per $j+1$

Terminazione ($j = n+1$):

- L'invariante dice che $A[1..n]$ è ordinato
- Questo è esattamente quello che volevamo dimostrare \square



Nota (Struttura di una dimostrazione con invariante).

1. Identificare l'invariante (cosa rimane vero ad ogni iterazione)
2. **Inizializzazione:** verificare prima del ciclo
3. **Mantenimento:** assumere vero all'iterazione k , dimostrare per $k + 1$
4. **Terminazione:** usare l'invariante + condizione di uscita per dimostrare la correttezza

4.0.12 Notazione Asintotica

La notazione asintotica serve per descrivere come cresce il tempo di esecuzione di un algoritmo quando l'input diventa molto grande.

$T(n)$: la complessità asintotica di un algoritmo:

- per n molto grandi
- a meno di costanti moltiplicative
- a meno di termini di ordine inferiore

Il costo sia in tempo che in spazio si descrivono in un ordine di grandezza.

Esempio 3.

$$T(n) = 3n^2 + 2n + 5 + \log n$$

$T(n)$ è una funzione quadratica in n , i termini di ordine inferiore vengono ignorati.

Esempio 4. $T(n) = 7n + 24$ è lineare in n

Esempio 5. $T(n) = 5$ è costante

Esempio 6. $T(n) = \log_3 n + 2$ è logaritmica in n

Nota. Gli ordini di grandezza sono insiemi infiniti che includono tutte le funzioni che, rispetto ad una data funzione $g(n)$, hanno un certo comportamento asintotico.

$$\begin{bmatrix} f(n) & \text{la "nostra" funzione} \\ g(n) & \text{la funzione di riferimento} \end{bmatrix}$$

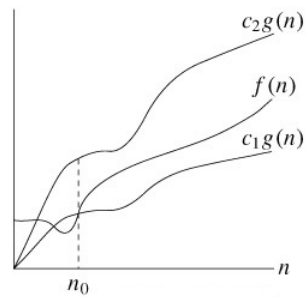
4.0.13 Notazione Θ - limite asintotico stretto

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

Si dice che $g(n)$ è un limite asintotico stretto per $f(n)$

Si scrive $f(n) \in \Theta(g(n))$ oppure $f(n) = \Theta(g(n))$

Si legge « $f(n)$ è in theta di $g(n)$ »



Nota. Al crescere di n , da un certo punto in poi (n_0), la funzione $f(n)$ è compresa in una fascia delimitata da $c_1g(n)$ e $c_2g(n)$

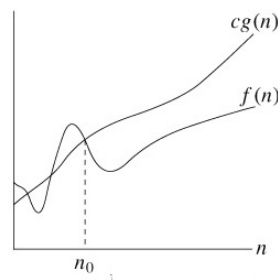
4.0.14 Notazione O - limite asintotico superiore

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

Si dice $g(n)$ è un limite asintotico superiore per $f(n)$

Si scrive $f(n) \in O(g(n))$ o $f(n) = O(g(n))$

Si legge « $f(n)$ è in O grande di $g(n)$ »



Nota. Al crescere di n , la funzione $f(n)$ non supera mai $cg(n)$

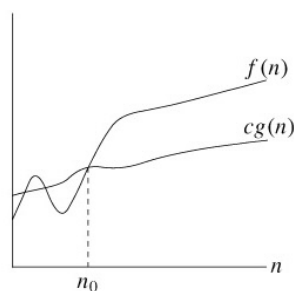
4.0.15 Notazione Ω - limite asintotico inferiore

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

Si dice $g(n)$ è un limite asintotico inferiore di $f(n)$

Si scrive $f(n) \in \Omega(g(n))$ o $f(n) = \Omega(g(n))$

Si legge « $f(n)$ è in Omega Grande di $g(n)$ »



4.0.16 Proprietà della notazione asintotica

Le notazioni Θ , O e Ω godono di alcune proprietà fondamentali che permettono di manipolare e confrontare le complessità degli algoritmi.

Definizione 4 (Transitività). Se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, allora $f(n) = \Theta(h(n))$

Questa proprietà vale anche per O e Ω :

- $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

Definizione 5 (Riflessività). Per ogni funzione $f(n)$:

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

Questa proprietà vale anche per O e Ω :

- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Definizione 6 (Simmetria).

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

Nota. La simmetria vale **solo** per Θ , non per O e Ω .

Definizione 7 (Simmetria trasposta).

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

Questa proprietà collega O e Ω : dire che f è limitata superiormente da g equivale a dire che g è limitata inferiormente da f .

Nota (Relazione tra le notazioni). Vale la seguente equivalenza:

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

Ovvero $f(n)$ ha limite asintotico stretto $g(n)$ se e solo se $g(n)$ è sia limite superiore che inferiore per $f(n)$.

4.0.17 Esercizi sulla notazione asintotica

Esempio 7 (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:

$$\forall n \geq n_0 : c_1 \cdot n^2 \leq 3n^2 - 2n - 1 \leq c_2 \cdot n^2$$

Limite superiore (O): Dimostriamo che $3n^2 - 2n - 1 \leq c_2 \cdot n^2$

Per $n \geq 1$: $3n^2 - 2n - 1 \leq 3n^2$

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: $3n^2 - 2n - 1 \geq c_1 \cdot n^2$

Dividendo per n^2 (per $n \geq 1$): $3 - \frac{2}{n} - \frac{1}{n^2} \geq c_1$

Per $n \geq 2$:

- $\frac{2}{n} \leq 1$
- $\frac{1}{n^2} \leq \frac{1}{4}$

Quindi: $3 - 1 - \frac{1}{4} = \frac{7}{4} \geq c_1$

Scegliendo $c_1 = 1$ (più semplice), 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 8 (Ordinamento per crescita asintotica). Ordinare le seguenti funzioni in ordine crescente di crescita asintotica:

$$2, \log n, \log^2 n, (\log n)^2, \sqrt{n}, n, n \log n, n \log^2 n, n^2, 2^n, 3^n, n!$$

Soluzione:

$$2 \prec \log n \prec \log^2 n = (\log n)^2 \prec \sqrt{n} \prec n \prec n \log n \prec n \log^2 n \prec n^2 \prec 2^n \prec 3^n \prec n!$$

Dove $f \prec g$ significa $f(n) = o(g(n))$, ovvero f cresce strettamente più lentamente di g .

Spiegazione delle classi:

- **Costanti:** $2 = \Theta(1)$
- **Logaritmiche:** $\log n, \log^2 n = (\log n)^2$
- **Sublineari:** $\sqrt{n} = n^{\frac{1}{2}}$
- **Lineari:** n
- **Linearitriche:** $n \log n, n \log^2 n$
- **Polinomiali:** n^2
- **Esponenziali:** $2^n \prec 3^n$ (base maggiore \Rightarrow crescita più rapida)
- **Fattoriali:** $n!$ (cresce più velocemente di qualsiasi esponenziale c^n)

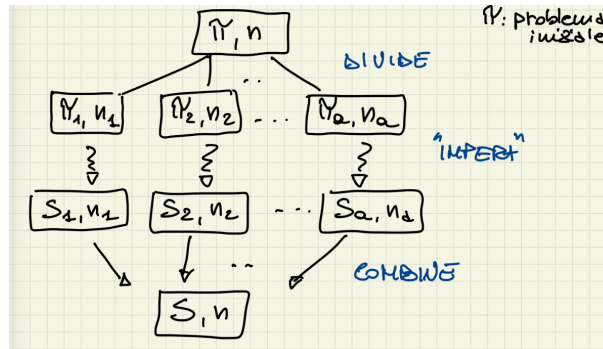
Nota. Per confrontare $\log^2 n$ e $(\log n)^2$: sono la stessa funzione! La notazione $\log^2 n$ significa $(\log n)^2$, non $\log(\log n)$.

5 Algoritmi di Ordinamento

5.0.1 Divide et Impera

Il Divide et Impera è un paradigma di programmazione adottato da molti algoritmi ricorsivi:

- Si divide il problema da risolvere in 2 o più sotto-problemi dello stesso tipo, ma che operano su un numero minore di dati (sottoinsieme di problema iniziale)
- Si risolvono i sottoproblemi in modo ricorsivo, con la stessa tecnica, o direttamente se si sono raggiunti i casi limite di dimensione minima del problema.
- Si combinano le soluzioni dei sottoproblemi per ottenere la soluzione del problema originale



5.0.2 Ricerca Binaria

5.0.3 Codice e complessità

Esempio 1 (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);
    }
}
```

La relazione di ricorrenza è

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(\frac{n}{2}) + \Theta(1) & n \geq 2 \end{cases}$$

$$f(n) = C(n) + D(n) = \text{combine} + \text{divide}$$

La complessità in tempo è $O(\log n)$

5.0.4 Correttezza

Tipicamente si dimostra per induzione sulla dimensione n del problema:

- **Caso base:** $n = 0, n = 1, \dots, n \leq n_0$ si dimostra che l'algoritmo è corretto nelle condizioni dei casi base in modo diretto.
- **Caso induttivo:** Si assume, per ipotesi induttiva, che le soluzioni dei sottoproblemi siano corrette (ovvero che l'algoritmo è corretto $\forall n' \ n_0 < n' < n$ e si dimostra che di conseguenza è corretto per n

5.0.5 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.

5.0.5.1 Ricerca Binaria Sinistra

Trova la **prima occorrenza** (più a sinistra) di un elemento k in un array ordinato.

Esempio 2 (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 è la prima occorrenza controllando che:

- $cx = sx$ (siamo al bordo sinistro), oppure
- $a[cx - 1] \neq k$ (l'elemento precedente è diverso)

Se non è la prima occorrenza, continuiamo a cercare nella metà sinistra.

5.0.5.2 Ricerca Binaria Destra

Trova l'**ultima occorrenza** (più a destra) di un elemento k in un array ordinato.

Esempio 3 (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);
    }
}
```

```

    }
    if (a[cx] <= k) {
        return ricercaBinariaDx(a, cx + 1, dx, k);
    } else {
        return ricercaBinariaDx(a, sx, cx - 1, k);
    }
}

```

Idea: quando troviamo k in posizione cx , verifichiamo se è l'ultima occorrenza controllando che:

- $cx = dx$ (siamo al bordo destro), oppure
- $a[cx + 1] \neq k$ (l'elemento successivo è diverso)

Se non è l'ultima occorrenza, continuiamo a cercare nella metà destra.

5.0.5.3 Conta Occorrenze

Usando le due varianti possiamo contare il numero di occorrenze di k in tempo $\Theta(\log n)$.

Esempio 4 (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;
}

```

Complessità: $\Theta(\log n)$

Eseguiamo al massimo due ricerche binarie, ciascuna con complessità $O(\log n)$, quindi la complessità totale è $\Theta(\log n)$.

Nota (Confronto con approccio lineare). Un approccio naive che scorre l'array contando le occorrenze richiede $\Theta(n)$. L'uso delle varianti della ricerca binaria permette di ottenere $\Theta(\log n)$, un miglioramento significativo per array di grandi dimensioni.

5.0.6 Merge Sort

Esempio 5 (Merge Sort).

```

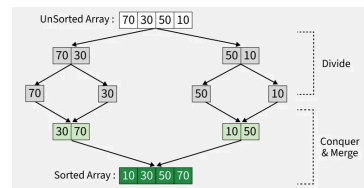
mergeSort(int[] A, int p, int r){           // -- T(n)
    if(p < r){                               // --  $\Theta(1)$ 
        int q = (p + r) / 2;                // divide --  $\Theta(1)$ 
        mergeSort(A, p, q);                 // impera --  $T(n/2)$ 
        mergeSort(A, q + 1, r);              // impera --  $T(n/2)$ 
        merge(A, p, q, r);                  // combine --  $\Theta(n)$ 
    }
}

```

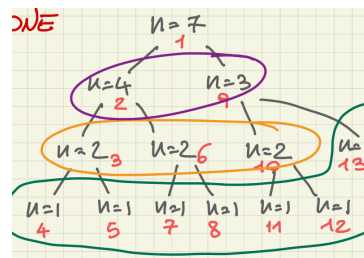
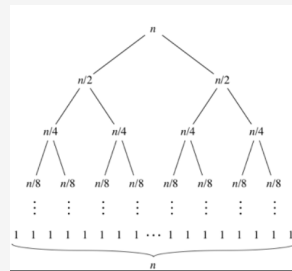
La relazione di ricorrenza dell'algoritmo è definita nel seguente modo:

$$T(n) = \begin{cases} \theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \theta(n) & n > 2 \end{cases}$$

La complessità in tempo in ogni caso è $O(n \log n)$



Nota. La relazione di ricorrenza è la definizione matematica di un processo, descrive il numero di operazioni di un algoritmo ricorsivo in funzione del suo input. La complessità in tempo invece è la misurazione asintotica di tale relazione



```
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. La complessità di Merge è lineare

Nota. In merge vengono utilizzati array di appoggio.

Per concludere la complessità in tempo del MergeSort è $\theta(n \log n)$ e la complessità in spazio è $O(n)$

5.0.7 Relazioni di ricorrenza

Le relazioni di ricorrenza servono a descrivere la complessità $T(n)$ di algoritmi ricorsivi.

- **Relazioni bilanciate:**

$$T(n) = \begin{cases} \theta(1) & n \leq n_0 \\ \underbrace{aT\left(\frac{n}{b}\right)}_{a \in \mathbb{N}^+ \wedge b > 1, b \in \mathbb{Q}} + \underbrace{f(n)}_{\text{forzante}} & n > n_0 \end{cases}$$

- **Relazioni di ordine k:**

$$T(n) = \begin{cases} \theta(1) & n \leq n_0 \\ \alpha_1 T(n-1) + \dots + \alpha_k T(n-k) & n > n_0 \end{cases}$$

Il caso generale è:

$$T(n) = \begin{cases} \theta(1) & n \leq n_0 \\ \alpha_1 T(n_1) + \dots + \alpha_k T(n-k) + f(n) & n > n_0 \end{cases}$$

Nota. Merge Sort e ricerca binaria sono relazioni bilanciate

5.0.8 Risoluzione relazioni di equivalenza

1. Metodo iterativo: si sviluppa la ricorrenza fino ai casi base
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
4. Teorema principale per le relazioni di ricorrenza bilanciate

5.0.9 Master Theorem

Con

$$T(n) = \begin{cases} \theta(1) & n \leq n_0 \\ aT(\frac{n}{b}) + f(n) & n > n_0 \end{cases}$$

Assumiamo che $a \geq 1$ e $b > 1$ e $f(n) > 0$, allora abbiamo che se:

1. $\exists \varepsilon > 0 : f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow T(n) = \theta(n^{\log_b a})$
2. $f(n) = \theta(n^{\log_b a}) \Rightarrow T(n) = \theta(n^{\log_b a} \times \log n)$
3. $\exists \varepsilon > 0 : f(n) = \Omega(n^{\log_b a + \varepsilon}), \exists c < 1 : af(\frac{n}{b}) \leq cf(n) \Rightarrow T(n) = \theta(f(n))$

Il secondo caso possiamo enunciarlo più generalmente nel seguente modo:

$$\exists k \geq 0 : f(n) = \theta(n^{\log_b a} \times \log^k n) \Rightarrow T(n) = \theta(n^{\log_b a} \times \log^{k+1} n)$$

5.0.9.1 Esempi di applicazione del Master Theorem

Esempio 6 (Caso 1: Ricerca binaria). $T(n) = T(\frac{n}{2}) + \Theta(1)$

Parametri: $a = 1, b = 2, f(n) = \Theta(1)$

Calcoliamo $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Confronto: $f(n) = \Theta(1) = \Theta(n^0)$

\Rightarrow **Caso 2:** $f(n) = \Theta(n^{\log_b a})$

Soluzione: $T(n) = \Theta(n^0 \cdot \log n) = \Theta(\log n)$

Esempio 7 (Caso 2: Merge Sort). $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Parametri: $a = 2, b = 2, f(n) = \Theta(n)$

Calcoliamo $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Confronto: $f(n) = \Theta(n) = \Theta(n^{\log_b a})$

\Rightarrow **Caso 2:** $f(n) = \Theta(n^{\log_b a})$

Soluzione: $T(n) = \Theta(n \cdot \log n) = \Theta(n \log n)$

Esempio 8 (Caso 1: Algoritmo ipotetico). $T(n) = 4T(\frac{n}{2}) + n$

Parametri: $a = 4, b = 2, f(n) = n$

Calcoliamo $n^{\log_b a} = n^{\log_2 4} = n^2$

Confronto: $f(n) = n = O(n^{2-\varepsilon})$ con $\varepsilon = 1$

\Rightarrow **Caso 1:** $f(n) = O(n^{\log_b a - \varepsilon})$

Soluzione: $T(n) = \Theta(n^2)$

Esempio 9 (Caso 3: Algoritmo ipotetico). $T(n) = T(\frac{n}{2}) + n$

Parametri: $a = 1, b = 2, f(n) = n$

Calcoliamo $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Confronto: $f(n) = n = \Omega(n^{0+\varepsilon})$ con $\varepsilon = 1$

Verifica condizione regolarità: $a \cdot f(\frac{n}{b}) = 1 \cdot \frac{n}{2} = \frac{n}{2} \leq c \cdot n$ per $c = \frac{1}{2}$

\Rightarrow **Caso 3:** $f(n) = \Omega(n^{\log_b a + \varepsilon})$

Soluzione: $T(n) = \Theta(n)$

Nota (Come scegliere il caso).

1. Calcola $n^{\log_b a}$
2. Confronta $f(n)$ con $n^{\log_b a}$:
 - $f(n)$ cresce **più lentamente** \rightarrow Caso 1
 - $f(n)$ cresce **allo stesso modo** \rightarrow Caso 2
 - $f(n)$ cresce **più velocemente** \rightarrow Caso 3 (verifica regolarità!)

5.0.10 Limiti inferiori alla difficoltà di un problema

Dato un problema π , la complessità al caso pessimo (in funzione della dimensione dell'input e asintotica) del miglior algoritmo che risolve π misura la difficoltà di π . Un algoritmo che risolve π fornisce un limite superiore alla difficoltà di π .

Definizione 1 (Limite inferiore per π). $L(n) \Rightarrow \forall$ algoritmo A che risolve π con $T_A(n) \in \Omega(L(n))$ (al caso pessimo). $L(n)$ è il numero minimo di operazioni che sono necessarie per risolvere π al caso peggiore

5.0.11 Criteri

5.0.11.1 Dimensione dell'input

Se la soluzione di un problema richiede l'esame di tutti i dati, allora la dimensione dell'input n è un limite inferiore: $L(n) = \Omega(n)$

Esempio 10. La ricerca max in un vettore non ordinato deve necessariamente analizzare tutti gli n elementi. ($L(n) = n$) (limite inferiore n).

Nota. La scansione lineare è un algoritmo che ha complessità lineare (limite superiore n). Un algoritmo di questo tipo è ottimo, e il limite inferiore lineare è significativo.

5.0.11.2 Albero di decisione

Criterio per stabilire un limite inferiore alla difficoltà di un problema π che si applica a problemi risolvibili attraverso una sequenza di «decisioni» (es. confronti tra valori) che via via riducono lo spazio delle soluzioni

Nota. Il caso pessimo di un algoritmo è la lunghezza max di radice-foglia (altezza dell'albero).

Nota. Se $SOL(n)$ è il numero di possibili soluzioni di π , allora $\log(SOL(n))$ è un limite inferiore per π

Il percorso radice-foglia è una possibile esecuzione: il percorso più breve è il caso ottimo; il percorso più lungo è il caso pessimo.

Nota. Un albero bilanciato è un albero che ha caso ottimo coincidente con il caso pessimo.

Definizione 2 (altezza albero). Altezza di un albero è la lunghezza del più lungo percorso dalla radice ad una foglia.

Nota. In un albero binario l'altezza minima $\geq \log_2(\#foglie)$

Definizione 3 (Albero bilanciato). albero bilanciato \Leftrightarrow altezza $\in \theta(\log n)$, $\theta(n)$ nodi e foglie

Nota. $\log(SOL(n)) = \log(\#foglie)$ è l'altezza minore che ci possa essere, ovvero è la complessità al caso pessimo del miglior algoritmo. L'algoritmo migliore al caso pessimo è quello che minimizza l'altezza dell'albero di decisione \Rightarrow è quello che ha altezza logaritmica rispetto al numero di foglie

5.0.11.3 Eventi Contabili

Se la ripetizione di un certo evento è indispensabile per risolvere π , allora ($\#$ volte che si deve ripetere \times costo evento) è un limite inferiore.

OSSERVAZIONE MY SOLUTION: SCANSIONE LINEARE

π : RICERCA K IN VETTORE NON ORDINATO

CRITERIO
ADD: $\#SOL = n+1 \Rightarrow \log n$ è un limite inferiore

HA ~~UN~~ UN ALG. CHE RISOLVA π IN $\log n$ AL CASO PESSIMO

NON È UN LIMITE INFERIORE SIGNIFICATIVO

CRITERIO
DIM. INPUT È APPLICABILE: DIM. INPUT n è un limite inferiore $\Rightarrow n$ è un limite inferiore

(1) SCANSIONE LINEARE È OTTIMA
(2) L.I. DIM. INPUT È SIGNIFICATIVO

5.0.12 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.

5.0.13 Idea dell'algoritmo

L'idea chiave è quella di scegliere un elemento detto **pivot** e partizionare l'array in modo che:

- tutti gli elementi minori o uguali al pivot siano a sinistra
- tutti gli elementi maggiori del pivot siano a destra

Successivamente si applica ricorsivamente lo stesso procedimento alle due partizioni.

5.0.14 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
        quickSort(A, q + 1, r);        // impera
    }
}
```

Nota. A differenza del MergeSort, nel QuickSort:

- Il lavoro principale è nella fase di **divide** (Partition)
- La fase di **combine** è banale (non serve fare nulla)

5.0.15 Procedura Partition

La procedura Partition riorganizza l'array $A[p..r]$ in loco, restituendo l'indice q tale che:

- $A[q]$ contiene il pivot
- $A[p..q-1]$ contiene elementi $\leq A[q]$
- $A[q+1..r]$ contiene elementi $> A[q]$

```
int partition(int[] A, int p, int r){
    int x = A[r];                // pivot (ultimo elemento)
    int i = p - 1;               // indice "bordo" partizione sinistra
    int j = p;
    while(j <= r - 1){
        if(A[j] <= x){
            i := i + 1;
            // swap(A[i], A[j])
            int temp = A[i];
            A[i] := A[j];
            A[j] := temp;
        }
        j := j + 1;
    }
    // swap(A[i+1], A[r]) - metti pivot in posizione finale
    int temp = A[i + 1];
    A[i + 1] := A[r];
    A[r] := temp;
    return i + 1;
}
```

5.0.16 Invariante di Partition

All'inizio di ogni iterazione del ciclo for:

1. Per ogni $k \in [p, i]$: $A[k] \leq x$ (pivot)
2. Per ogni $k \in [i + 1, j - 1]$: $A[k] > x$
3. $A[r] = x$ (il pivot)

5.0.17 Correttezza di Partition

Dimostrazione. Si dimostra per induzione sul numero di iterazioni:

- **Caso base** ($j = p$): le regioni $[p, i]$ e $[i + 1, j - 1]$ sono vuote, l'invariante è banalmente vero.
- **Passo induttivo:** assumiamo l'invariante vero prima dell'iterazione j -esima.
 - Se $A[j] > x$: incrementiamo solo j , la regione degli elementi $> x$ si estende
 - Se $A[j] \leq x$: incrementiamo i , scambiamo $A[i]$ con $A[j]$, la regione degli elementi $\leq x$ si estende

■

5.0.18 Complessità di Partition

La procedura Partition ha complessità $\Theta(n)$ dove $n = r - p + 1$ è il numero di elementi da partizionare.

5.0.19 Analisi di QuickSort

La complessità di QuickSort dipende dal bilanciamento delle partizioni.

5.0.20 Caso Pessimo

Il caso pessimo si verifica quando le partizioni sono sempre massimamente sbilanciate:

- Una partizione ha $n - 1$ elementi
- L'altra ha 0 elementi

Questo accade quando l'array è già ordinato (o inversamente ordinato) e scegliamo sempre il primo o l'ultimo elemento come pivot.

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

Risolvendo la ricorrenza:

$$T(n) = \sum_{k=1}^n \Theta(k) = \Theta(n^2)$$

Nota. Nel caso pessimo QuickSort ha la stessa complessità degli algoritmi quadratici come Insertion Sort!

5.0.21 Caso Ottimo

Il caso ottimo si verifica quando le partizioni sono sempre perfettamente bilanciate:

- Ogni partizione ha circa $\frac{n}{2}$ elementi

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

Per il Master Theorem (caso 2 con $a = 2$, $b = 2$, $f(n) = \Theta(n)$):

$$T(n) = \Theta(n \log n)$$

5.0.22 Caso Medio

Teorema 1 (Complessità media di QuickSort). Se tutte le permutazioni dell'input sono equiprobabili, la complessità attesa di QuickSort è $\Theta(n \log n)$.

L'intuizione è che anche partizioni moderatamente sbilanciate (es. 9:1) producono alberi di ricorsione con altezza $O(\log n)$.

5.0.23 QuickSort Randomizzato

Per evitare il caso pessimo su input «cattivi», si può randomizzare la scelta del pivot.

```
int randomizedPartition(int[] A, int p, int r){
    int i = random(p, r);    // scegli pivot casuale
    // swap(A[i], A[r]) - mettilo in fondo
    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);
    }
}
```

Definizione 4 (Complessità attesa). Il numero atteso di confronti di RandomizedQuickSort è $O(n \log n)$, indipendentemente dall'input.

5.0.24 Analisi del numero atteso di confronti

Siano z_1, z_2, \dots, z_n gli elementi di A in ordine crescente.

Definiamo X_{ij} come variabile indicatrice:

$$X_{ij} = \begin{cases} 1 & \text{se } z_i \text{ e } z_j \text{ vengono confrontati} \\ 0 & \text{altrimenti} \end{cases}$$

Il numero totale di confronti è:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

Teorema 2 (Probabilità di confronto).

$$P(X_{ij} = 1) = \frac{2}{j - i + 1}$$

Questo perché z_i e z_j vengono confrontati se e solo se uno dei due è il primo elemento scelto come pivot nell'insieme $\{z_i, z_{i+1}, \dots, z_j\}$.

Quindi:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = O(n \log n)$$

5.0.25 Moltiplicazione veloce di interi

Un'applicazione importante del Divide et Impera è la moltiplicazione di interi di lunghezza arbitraria.

5.0.26 Metodo tradizionale

La moltiplicazione «delle elementari» di due numeri di n cifre richiede $\Theta(n^2)$ operazioni.

5.0.27 Algoritmo di Karatsuba

Siano X e Y due numeri di n cifre (assumiamo n potenza di 2). Scriviamo:

$$X = X_1 \cdot 10^{\frac{n}{2}} + X_0$$

$$Y = Y_1 \cdot 10^{\frac{n}{2}} + Y_0$$

dove X_1, X_0, Y_1, Y_0 hanno $\frac{n}{2}$ cifre.

Il prodotto $X \cdot Y$ è:

$$X \cdot Y = X_1 Y_1 \cdot 10^n + (X_1 Y_0 + X_0 Y_1) \cdot 10^{\frac{n}{2}} + X_0 Y_0$$

Questo richiede 4 moltiplicazioni di numeri di $\frac{n}{2}$ cifre.

Nota. L'idea di Karatsuba è ridurre le moltiplicazioni da 4 a 3:

$$X_1 Y_0 + X_0 Y_1 = (X_1 + X_0)(Y_1 + Y_0) - X_1 Y_1 - X_0 Y_0$$

5.0.28 Algoritmo

```
int karatsuba(int X, int Y, int n){
    if(n <= n0){
        return X * Y; // moltiplicazione diretta
    }

    // Dividi
    int X1 = X / pow(10, n/2);
    int X0 = X % pow(10, n/2);
    int Y1 = Y / pow(10, n/2);
    int Y0 = Y % pow(10, n/2);

    // Impera (3 moltiplicazioni ricorsive)
    int P1 = karatsuba(X1, Y1, n/2);
    int P2 = karatsuba(X0, Y0, n/2);
    int P3 = karatsuba(X1 + X0, Y1 + Y0, n/2);

    // Combina
    return P1 * pow(10, n) + (P3 - P1 - P2) * pow(10, n/2) + P2;
}
```

5.0.29 Complessità

La relazione di ricorrenza è:

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

Per il Master Theorem (caso 1 con $a = 3$, $b = 2$, $\log_2 3 \approx 1.585$):

$$T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

Nota. Il miglioramento da n^2 a $n^{1.585}$ è significativo per numeri molto grandi (es. crittografia RSA).

5.0.30 Confronto algoritmi di ordinamento

Algoritmo	Caso Ottimo	Caso Medio	Caso Pessimo
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
QuickSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
QuickSort Rand.	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)^*$

Tabella 9: * atteso

5.0.31 Heap e Heapsort

5.0.32 Definizione di Heap

Definizione 5 (Heap). Un **heap** (binario) è un albero binario quasi completo che soddisfa la **proprietà di heap**:

- **Max-Heap:** per ogni nodo i diverso dalla radice: $A[\text{Parent}(i)] \geq A[i]$
- **Min-Heap:** per ogni nodo i diverso dalla radice: $A[\text{Parent}(i)] \leq A[i]$

Definizione 6 (Albero binario quasi completo). Un albero binario è **quasi completo** se tutti i livelli sono completamente riempiti, eccetto eventualmente l'ultimo che è riempito da sinistra a destra.

5.0.33 Proprietà

- L'elemento massimo (in un max-heap) è sempre nella radice
- L'altezza di un heap con n elementi è $\Theta(\log n)$
- Un heap con n elementi ha al massimo $\lceil \frac{n}{2^{h+1}} \rceil$ nodi a altezza h

5.0.34 Realizzazione implicita come Array

Un heap può essere rappresentato efficientemente come un array A dove:

- $A[1]$ è la radice
- Per un nodo in posizione i :
 - **Parent:** $\lfloor \frac{i}{2} \rfloor$
 - **Left child:** $2i$
 - **Right child:** $2i + 1$

```
int parent(int i){ return i / 2; }
int left(int i) { return 2 * i; }
int right(int i) { return 2 * i + 1; }
```

Nota. Questa rappresentazione è molto efficiente:

- Non servono puntatori (risparmio di spazio)
- Navigazione padre/figli in $O(1)$
- Cache-friendly (elementi consecutivi in memoria)

Esempio 11. Array: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Rappresenta un max-heap dove:

- Radice = 16
- Figli di 16: 14 (sinistro), 10 (destro)
- Figli di 14: 8, 7
- Figli di 10: 9, 3
- ...

5.0.35 Max-Heapify

La procedura Max-Heapify ripristina la proprietà di max-heap su un sottoalbero, assumendo che i sottoalberi sinistro e destro siano già max-heap.

```
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 è il nodo corrente, scambia e ricorri
    if(largest != i){
        int temp = A[i];
        A[i] := A[largest];
        A[largest] := temp;
        maxHeapify(A, largest, heapsize);
    }
}
```

5.0.36 Correttezza

Dimostrazione. Per induzione sull'altezza del nodo i :

- **Caso base** (altezza 0, foglia): non c'è nulla da fare
- **Passo induttivo**: se $A[i]$ viola la proprietà, lo scambiamo con il figlio maggiore. Dopo lo scambio, il sottoalbero radicato in quel figlio potrebbe violare la proprietà, ma ha altezza minore.

■

5.0.37 Complessità

$$T(n) = O(h) = O(\log n)$$

dove h è l'altezza del nodo i . Nel caso peggior, l'elemento «scende» fino alle foglie.

Esempio 12 (Max-Heapify passo-passo). Consideriamo l'array $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$ e chiamiamo $\text{Max-Heapify}(A, 2)$ (indice del nodo con valore 4).

I sottoalberi sinistro (radice 14) e destro (radice 7) sono già max-heap, ma il nodo 4 viola la proprietà.

Passo 1: Confronto iniziale

- Nodo corrente: $A[2] = 4$
- Figlio sinistro: $A[4] = 14$
- Figlio destro: $A[5] = 7$
- Il massimo tra $\{4, 14, 7\}$ è 14 (in posizione 4)
- Poiché $14 > 4$, scambiamo $A[2]$ con $A[4]$

Array dopo passo 1: $[16, 14, 10, 4, 7, 9, 3, 2, 8, 1]$

Passo 2: Ricorsione su posizione 4

- Nodo corrente: $A[4] = 4$
- Figlio sinistro: $A[8] = 2$
- Figlio destro: $A[9] = 8$
- Il massimo tra $\{4, 2, 8\}$ è 8 (in posizione 9)
- Poiché $8 > 4$, scambiamo $A[4]$ con $A[9]$

Array dopo passo 2: $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$

Passo 3: Ricorsione su posizione 9

- Nodo corrente: $A[9] = 4$
- Figlio sinistro: $A[18]$ non esiste (fuori dall'heap)
- Figlio destro: $A[19]$ non esiste (fuori dall'heap)
- Il nodo è una foglia: terminazione

Risultato finale: $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$

Il valore 4 è «sceso» dalla posizione 2 alla posizione 9, attraverso 2 scambi.

5.0.38 Build-Max-Heap

Costruisce un max-heap a partire da un array non ordinato.

```
buildMaxHeap(int[] A, int n){
    int heapsize = n;
    int i = n / 2;
    while(i >= 1){
        maxHeapify(A, i, heapsize);
        i := i - 1;
    }
}
```

Nota. Partiamo da $\lfloor \frac{n}{2} \rfloor$ perché i nodi da $\lfloor \frac{n}{2} \rfloor + 1$ a n sono foglie e sono già heap (banalmente).

5.0.39 Correttezza

Definizione 7 (Invariante di ciclo). All'inizio di ogni iterazione del ciclo `for`, ogni nodo $i + 1, i + 2, \dots, n$ è radice di un max-heap.

Dimostrazione.

- **Inizializzazione:** Prima della prima iterazione ($i = \lfloor \frac{n}{2} \rfloor$), i nodi $\lfloor \frac{n}{2} \rfloor + 1, \dots, n$ sono foglie, quindi banalmente radici di max-heap.
- **Mantenimento:** $\text{Max-Heapify}(A, i)$ rende i radice di un max-heap (i figli sono già radici di max-heap per l'invariante)
- **Terminazione:** Quando $i = 0$, ogni nodo $1, 2, \dots, n$ è radice di un max-heap. In particolare il nodo 1 (radice).

■

Esempio 13 (Build-Max-Heap passo-passo). Costruiamo un max-heap dall'array $A = [7, 4, 3, 8, 9, 6]$ con $n = 6$.

Calcoliamo $\lfloor n/2 \rfloor = \lfloor 6/2 \rfloor = 3$, quindi partiamo da $i = 3$ e andiamo fino a $i = 1$.

Stato iniziale:

Array: $[7, 4, 3, 8, 9, 6]$ (indici 1-6)

Iterazione 1: Max-Heapify(A, 3)

- Nodo: $A[3] = 3$
- Figlio sinistro: $A[6] = 6$
- Figlio destro: non esiste
- Massimo: $6 > 3$, scambio $A[3]$ con $A[6]$

Array dopo $i=3$: $[7, 4, 6, 8, 9, 3]$

Iterazione 2: Max-Heapify(A, 2)

- Nodo: $A[2] = 4$
- Figlio sinistro: $A[4] = 8$
- Figlio destro: $A[5] = 9$
- Massimo: $9 > 4$, scambio $A[2]$ con $A[5]$
- Ricorsione su posizione 5: è una foglia, termina

Array dopo $i=2$: $[7, 9, 6, 8, 4, 3]$

Iterazione 3: Max-Heapify(A, 1)

- Nodo: $A[1] = 7$
- Figlio sinistro: $A[2] = 9$
- Figlio destro: $A[3] = 6$
- Massimo: $9 > 7$, scambio $A[1]$ con $A[2]$

Array intermedio: $[9, 7, 6, 8, 4, 3]$

- Ricorsione su posizione 2:
 - Nodo: $A[2] = 7$
 - Figlio sinistro: $A[4] = 8$
 - Figlio destro: $A[5] = 4$
 - Massimo: $8 > 7$, scambio $A[2]$ con $A[4]$

Array dopo $i=1$: $[9, 8, 6, 7, 4, 3]$

- Ricorsione su posizione 4: è una foglia, termina

Risultato finale: $[9, 8, 6, 7, 4, 3]$

Iterazione	i	Operazione	Array
Iniziale	-	-	$[7, 4, 3, 8, 9, 6]$

1	3	scambio 3 ↔ 6	[7, 4, 6, 8, 9, 3]
2	2	scambio 4 ↔ 9	[7, 9, 6, 8, 4, 3]
3	1	scambio 7 ↔ 9, poi 7 ↔ 8	[9, 8, 6, 7, 4, 3]

5.0.40 Complessità

5.0.40.1 Analisi naive

$O(n)$ chiamate a Max-Heapify, ognuna costa $O(\log n)$, quindi $O(n \log n)$.

5.0.40.2 Analisi più stretta

Teorema 3 (Complessità di Build-Max-Heap). Build-Max-Heap ha complessità $\Theta(n)$.

Dimostrazione. Un heap di n elementi ha altezza $h = \lfloor \log n \rfloor$.

A ogni altezza i , ci sono al massimo $\lceil \frac{n}{2^{i+1}} \rceil$ nodi.

Il costo di Max-Heapify su un nodo a altezza i è $O(i)$.

Costo totale:

$$\sum_{i=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{i+1}} \right\rceil \cdot O(i) = O\left(n \sum_{i=0}^{\lfloor \log n \rfloor} \frac{i}{2^i}\right)$$

Usando $\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$ (serie convergente):

$$O(n \cdot 2) = O(n)$$

■

5.0.41 Heapsort

L'algoritmo Heapsort sfrutta la struttura heap per ordinare un array.

```
heapsort(int[] A, int n){
    buildMaxHeap(A, n);
    int heapsize = n;
    int i = n;
    while(i >= 2){
        // swap(A[1], A[i]) - metti il max in fondo
        int temp = A[1];
        A[1] := A[i];
        A[i] := temp;

        heapsize := heapsize - 1;
        maxHeapify(A, 1, heapsize); // ripristina heap
        i := i - 1;
    }
}
```

5.0.42 Idea

1. Costruisci un max-heap dall'array
2. Il massimo è in $A[1]$: scambialo con l'ultimo elemento

3. Riduci la dimensione dell'heap di 1
4. Ripristina la proprietà di max-heap sulla radice
5. Ripeti finché l'heap ha un solo elemento

5.0.43 Correttezza

Definizione 8 (Invariante di ciclo). All'inizio di ogni iterazione:

- $A[1..i]$ è un max-heap contenente gli i elementi più piccoli
- $A[i+1..n]$ contiene gli $n-i$ elementi più grandi, ordinati

5.0.44 Complessità

$$T(n) = \underbrace{O(n)}_{\text{Build-Max-Heap}} + \underbrace{(n-1) \cdot O(\log n)}_{\text{ciclo for}} = O(n \log n)$$

Nota. Heapsort:

- Ha complessità $O(n \log n)$ in **tutti** i casi
- È **in-place** (non richiede memoria aggiuntiva)
- Non è stabile (può cambiare l'ordine relativo di elementi uguali)

Esempio 14 (HeapSort passo-passo). Ordiniamo l'array $A = [7, 4, 3, 8, 9, 6]$ usando HeapSort.

Fase 1: Build-Max-Heap

Come visto nell'esempio precedente, dopo Build-Max-Heap otteniamo:

$$A = [9, 8, 6, 7, 4, 3] \text{ con heapsize} = 6$$

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 ordinata (destra).

Dettaglio iterazione 1:

- Scambio $A[1] = 9$ con $A[6] = 3$: array = $[3, 8, 6, 7, 4, 9]$
- Riduco heapsize a 5 (il 9 è «fuori» dall'heap, nella sua posizione finale)
- Max-Heapify($A, 1, 5$):
 - $A[1] = 3$, figli: $A[2] = 8, A[3] = 6$
 - Massimo: 8, scambio $A[1]$ con $A[2]$: $[8, 3, 6, 7, 4, 9]$
 - Ricorsione su posizione 2: $A[2] = 3$, figli: $A[4] = 7, A[5] = 4$
 - Massimo: 7, scambio $A[2]$ con $A[4]$: $[8, 7, 6, 3, 4, 9]$

Dettaglio iterazione 2:

- 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 $A[1]$ con $A[2]$: $[7, 4, 6, 3, 8, 9]$
- Ricorsione su posizione 2: $A[2] = 4$, figli: $A[4] = 3$, nessun figlio destro
- $4 > 3$: nessuno scambio, termina

Dettaglio iterazione 3:

- 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 $A[1]$ con $A[3]$: $[6, 4, 3, 7, 8, 9]$
 - Ricorsione su posizione 3: è una foglia, termina

Dettaglio iterazione 4:

- 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: $A[2] = 4$
 - Massimo: 4, scambio $A[1]$ con $A[2]$: $[4, 3, 6, 7, 8, 9]$

Dettaglio iterazione 5:

- 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: $[3, 4, 6, 7, 8, 9]$ (array ordinato in ordine crescente)

5.0.45 Code di Priorità

Definizione 9 (Coda di priorità). Una **coda di priorità** è una struttura dati che mantiene un insieme S di elementi, ognuno con una **chiave** (priorità) associata, e supporta le seguenti operazioni:

- **Insert**(S , x): inserisce x in S
- **Maximum**(S): restituisce l'elemento con chiave massima
- **Extract-Max**(S): rimuove e restituisce l'elemento con chiave massima
- **Increase-Key**(S , x , k): aumenta la chiave di x al nuovo valore k

5.0.46 Applicazioni

- Scheduling di processi (priorità = importanza del processo)
- Simulazione di eventi discreti (priorità = tempo dell'evento)
- Algoritmo di Dijkstra (priorità = distanza stimata)
- Algoritmo di Prim per MST

5.0.47 Implementazione con Max-Heap**5.0.47.1 Maximum**

```
int heapMaximum(int[] A){
    return A[1];
}
```

Complessità: $O(1)$

5.0.47.2 Extract-Max

```

int heapExtractMax(int[] A, int heapsize){
    if(heapsize < 1){
        // error "heap underflow"
        return -1;
    }
    int max = A[1];
    A[1] := A[heapsize];
    heapsize := heapsize - 1;
    maxHeapify(A, 1, heapsize);
    return max;
}

```

Complessità: $O(\log n)$

5.0.47.3 Increase-Key

```

heapIncreaseKey(int[] A, int i, int key){
    if(key < A[i]){
        // error "nuova chiave minore della precedente"
        return;
    }
    A[i] := key;
    // Risali verso la radice finché 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);
    }
}

```

Complessità: $O(\log n)$

5.0.47.4 Insert

```

heapInsert(int[] A, int key, int heapsize){
    heapsize := heapsize + 1;
    A[heapsize] :=  $-\infty$ ;
    heapIncreaseKey(A, heapsize, key);
}

```

Complessità: $O(\log n)$

5.0.48 Riepilogo complessità

Operazione	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 10: Confronto implementazioni code di priorità

Nota. Lo heap offre un buon compromesso: tutte le operazioni hanno costo logaritmico, mentre con un array non ordinato alcune operazioni sono costanti ma **Maximum** ed **Extract-Max** richiedono tempo lineare.

5.0.49 Ordinamento in tempo lineare

5.0.50 Limite inferiore per l'ordinamento per confronti

Teorema 4 (Limite inferiore). Qualsiasi algoritmo di ordinamento basato su confronti richiede $\Omega(n \log n)$ confronti nel caso pessimo.

5.0.51 Dimostrazione con albero di decisione

Definizione 10 (Albero di decisione). Un **albero di decisione** è un albero binario che rappresenta i confronti effettuati da un algoritmo di ordinamento su un input di dimensione n :

- Ogni nodo interno rappresenta un confronto $a_i \leq a_j$
- Il sottoalbero sinistro corrisponde al caso «sì»
- Il sottoalbero destro corrisponde al caso «no»
- Ogni foglia rappresenta una permutazione dell'output

Dimostrazione.

- Un array di n elementi può avere $n!$ permutazioni
- L'albero di decisione deve avere almeno $n!$ foglie (una per ogni possibile output)
- Un albero binario di altezza h ha al massimo 2^h foglie
- Quindi: $2^h \geq n! \Rightarrow h \geq \log(n!)$

Usando l'approssimazione di Stirling: $\log(n!) = \Theta(n \log n)$

Quindi: $h = \Omega(n \log n)$ ■

Nota. Questo limite inferiore ci dice che **non possiamo fare meglio** di $O(n \log n)$ usando solo confronti. MergeSort e HeapSort sono quindi **ottimi** tra gli algoritmi basati su confronti.

5.0.52 Ordinamento in tempo lineare

Per superare il limite $\Omega(n \log n)$, dobbiamo usare algoritmi che **non** si basano esclusivamente su confronti. Questi algoritmi sfruttano informazioni aggiuntive sui dati.

5.0.53 Counting Sort

Il Counting Sort funziona quando gli elementi da ordinare sono interi in un intervallo noto $[0, k]$.

5.0.54 Idea

1. Conta quante volte appare ogni valore
2. Usa questi conteggi per determinare la posizione finale di ogni elemento

5.0.55 Algoritmo

```
countingSort(int[] A, int[] B, int n, int k){
    // C[i] conterrà il numero di elementi <= i
    int[] C = new int[k + 1];

    // Inizializza C a 0
    int i = 0;
    while(i <= k){
        C[i] := 0;
        i := i + 1;
    }
}
```

```

}

// C[i] = numero di elementi uguali a i
int j = 1;
while(j <= n){
    C[A[j]] := C[A[j]] + 1;
    j := j + 1;
}

// C[i] = numero di elementi <= i
i := 1;
while(i <= k){
    C[i] := C[i] + C[i - 1];
    i := i + 1;
}

// Costruisci l'array ordinato B
j := n;
while(j >= 1){
    B[C[A[j]]] := A[j];
    C[A[j]] := C[A[j]] - 1;
    j := j - 1;
}
}

```

Esempio 15 (Counting Sort passo-passo). Input: $A = [2, 5, 3, 0, 2, 3, 0, 3]$ con $n = 8$, $k = 5$

Fase 1: Inizializzazione di C (array di conteggio)

$$C = [0, 0, 0, 0, 0, 0] \quad (\text{indici } 0..5)$$

Fase 2: Conteggio delle occorrenze di ogni valore

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]

Interpretazione: 2 zeri, 0 uni, 2 due, 3 tre, 0 quattro, 1 cinque

Fase 3: Somme prefisse ($C[i] = \text{quanti elementi} \leq i$)

i	Calcolo	C
0	già ok	[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]

Fase 4: Costruzione output (da destra a sinistra per stabilità)

j	A[j]	C[A[j]]	$B[C[A[j]]] := A[j]$	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: $B = [0, 0, 2, 2, 3, 3, 3, 5]$ ✓

Nota (Perché si scorre da destra a sinistra?). Scorrendo A da destra a sinistra, gli elementi con lo stesso valore vengono inseriti in B mantenendo l'ordine relativo originale. Questo rende Counting Sort **stabile**, proprietà essenziale per Radix Sort.

5.0.56 Complessità

- Inizializzazione di C : $\Theta(k)$
- Conteggio elementi: $\Theta(n)$
- Somme prefisse: $\Theta(k)$
- Costruzione output: $\Theta(n)$

$$T(n, k) = \Theta(n + k)$$

Nota. Se $k = O(n)$, allora $T(n) = \Theta(n)$ - ordinamento lineare!

5.0.57 Stabilità

Definizione 11 (Algoritmo stabile). Un algoritmo di ordinamento è **stabile** se elementi con la stessa chiave mantengono l'ordine relativo che avevano nell'input.

Nota. Counting Sort è **stabile** (grazie al ciclo finale che scorre da destra a sinistra). Questa proprietà è fondamentale per Radix Sort.

5.0.58 Limitazioni

- Richiede che gli elementi siano interi non negativi
- Richiede di conoscere il range $[0, k]$
- Se $k \gg n$, lo spazio e il tempo diventano proibitivi

5.0.59 Radix Sort

Il Radix Sort ordina numeri interi cifra per cifra, dalla meno significativa alla più significativa.

5.0.60 Idea

Ordina ripetutamente usando un algoritmo stabile (come Counting Sort) su ogni cifra, partendo dalla cifra meno significativa.

5.0.61 Algoritmo

```
radixSort(int[] A, int d){
    // d = numero di cifre
    int i = 1;
    while(i <= d){
        // Usa un ordinamento stabile sulla cifra i-esima
        stableSort(A, i);
        i := i + 1;
    }
}
```

Esempio 16 (Radix Sort passo-passo). Input: $A = [329, 457, 657, 839, 436, 720, 355]$, $d = 3$ cifre

Passo 1: Ordinamento per UNITÀ (cifra meno significativa)

Numero	Cifra unità	Gruppo
329	9	9
457	7	7
657	7	7
839	9	9
436	6	6
720	0	0
355	5	5

Ordinando stabilmente per unità: $[720, 355, 436, 457, 657, 329, 839]$

Passo 2: Ordinamento per DECINE

Numero	Cifra decine	Gruppo
720	2	2
355	5	5
436	3	3
457	5	5
657	5	5
329	2	2
839	3	3

Ordinando stabilmente per decine: $[720, 329, 436, 839, 355, 457, 657]$

Nota: 720 viene prima di 329 (entrambi hanno decine=2), perché 720 era prima nell'array precedente → **stabilità!**

Passo 3: Ordinamento per CENTINAIA

Numero	Cifra centinaia	Gruppo
720	7	7
329	3	3
436	4	4
839	8	8
355	3	3
457	4	4
657	6	6

Ordinando stabilmente per centinaia: [329, 355, 436, 457, 657, 720, 839] ✓

Nota: 329 viene prima di 355 (entrambi hanno centinaia=3), perché 329 era prima dopo il passo 2 → **stabilità!**

Nota (Importanza della stabilità in Radix Sort). Se l'ordinamento su ogni cifra non fosse stabile, il lavoro fatto sulle cifre precedenti andrebbe perso. Ad esempio, dopo aver ordinato per unità e decine, 329 e 355 sono correttamente in quest'ordine (perché $29 < 55$). L'ordinamento stabile per centinaia preserva questo ordine tra elementi con la stessa cifra delle centinaia.

Nota. È fondamentale usare un ordinamento **stabile**! Altrimenti l'ordinamento sulle cifre più significative distruggerebbe il lavoro fatto sulle cifre meno significative.

5.0.62 Correttezza

Dimostrazione. Per induzione sul numero di cifre ordinate:

- Dopo l'ordinamento sulla cifra 1, gli elementi sono ordinati rispetto a quella cifra
- Assumiamo che dopo $i - 1$ iterazioni, gli elementi siano ordinati rispetto alle ultime $i - 1$ cifre
- L'ordinamento stabile sulla cifra i mantiene l'ordine relativo degli elementi con la stessa cifra i , che erano già ordinati rispetto alle cifre $1, \dots, i - 1$

■

5.0.63 Complessità

Se usiamo Counting Sort come algoritmo stabile, con numeri in base b :

- Ogni cifra è in $[0, b - 1]$
- Counting Sort su ogni cifra: $\Theta(n + b)$
- Numero di cifre: $d = \log_{b(\max)}$

$$T(n) = \Theta(d(n + b))$$

Nota. Se $b = n$ e d è costante, allora $T(n) = \Theta(n)$ - ordinamento lineare!

5.0.64 Come scegliere la base

Dato un numero massimo con r bit, possiamo scegliere la base $b = 2^s$ dove s divide r .

Teorema 5 (Scelta ottimale della base). Per n numeri a r bit, con $r \geq \log n$, Radix Sort ordina in tempo $\Theta(n)$ scegliendo $s = \log n$ (quindi $b = n$).

5.0.65 Bucket Sort

Il Bucket Sort assume che l'input sia distribuito uniformemente in un intervallo.

5.0.66 Idea

1. Dividi l'intervallo in n «bucket» (secchi) di uguale dimensione
2. Distribuisci gli elementi nei bucket
3. Ordina ogni bucket (con Insertion Sort)
4. Concatena i bucket

5.0.67 Algoritmo

```
bucketSort(int[] A, int n){
    // B = array di n liste vuote
    List[] B = new List[n];

    // Distribuisci gli elementi nei bucket
    int i = 1;
    while(i <= n){
        int idx = n * A[i]; // floor implicito
        insert(B[idx], A[i]);
        i := i + 1;
    }

    // Ordina ogni bucket
    i := 0;
    while(i <= n - 1){
        insertionSort(B[i]);
        i := i + 1;
    }

    // Concatena i bucket
    return concatenate(B[0], B[1], ..., B[n-1]);
}
```

Nota. Assumiamo che gli elementi siano in $[0, 1)$. Per altri intervalli, si normalizza.

5.0.68 Complessità

- Caso pessimo: $\Theta(n^2)$ (tutti gli elementi nello stesso bucket)
- Caso medio (distribuzione uniforme): $\Theta(n)$

Teorema 6 (Complessità attesa). Se l'input è distribuito uniformemente in $[0, 1)$, il tempo atteso di Bucket Sort è $\Theta(n)$.

5.0.69 Riepilogo algoritmi di ordinamento

Algoritmo	Caso Pessimo	Caso Medio	Stabile?	In-place?
Insertion Sort	$O(n^2)$	$O(n^2)$	Sì	Sì
Selection Sort	$O(n^2)$	$O(n^2)$	No	Sì
Merge Sort	$O(n \log n)$	$O(n \log n)$	Sì	No
QuickSort	$O(n^2)$	$O(n \log n)$	No	Sì
Heapsort	$O(n \log n)$	$O(n \log n)$	No	Sì
Counting Sort	$O(n + k)$	$O(n + k)$	Sì	No
Radix Sort	$O(d(n + b))$	$O(d(n + b))$	Sì	No
Bucket Sort	$O(n^2)$	$O(n)$	Sì	No

Tabella 17: Confronto algoritmi di ordinamento

Nota.

- Gli algoritmi basati su confronti hanno limite inferiore $\Omega(n \log n)$
- Counting Sort, Radix Sort e Bucket Sort possono essere lineari ma richiedono ipotesi aggiuntive sull'input
- La scelta dell'algoritmo dipende dalle caratteristiche dei dati e dai vincoli di spazio

6 Strutture Dati

6.0.1 Strutture Dati Lineari

Le strutture dati lineari sono collezioni di elementi organizzati in sequenza, dove ogni elemento (tranne il primo e l'ultimo) ha un predecessore e un successore.

6.0.2 Array

Definizione 1 (Array). Un **array** è una struttura dati che memorizza elementi dello stesso tipo in locazioni di memoria **contigue**. Ogni elemento è identificato da un **indice**.

6.0.3 Proprietà

- Accesso diretto in $O(1)$ tramite indice
- Dimensione fissa (in molti linguaggi)
- Elementi contigui in memoria (cache-friendly)

6.0.4 Operazioni e complessità

Operazione	Descrizione	Complessità
Access(A, i)	Accedi all'elemento in posizione i	$O(1)$
Search(A, x)	Cerca l'elemento x	$O(n)$
Insert(A, i, x)	Inserisci x in posizione i	$O(n)$
Delete(A, i)	Elimina l'elemento in posizione i	$O(n)$

Tabella 18: Operazioni su array

Nota. L'inserimento e la cancellazione richiedono lo spostamento degli elementi successivi, da cui la complessità lineare.

6.0.5 Liste

Definizione 2 (Lista concatenata). Una **lista concatenata** è una struttura dati in cui gli elementi (nodi) sono collegati tramite **puntatori**. Ogni nodo contiene:

- Un **dato** (o chiave)
- Uno o più **puntatori** ad altri nodi

6.0.6 Lista semplicemente concatenata

Ogni nodo contiene un puntatore al nodo successivo.

```
// Struttura Nodo (lista semplice)
// key : dato
// next : puntatore al nodo successivo
```

6.0.7 Lista doppiamente concatenata

Ogni nodo contiene puntatori sia al successore che al predecessore.

```
// Struttura Nodo (lista doppia)
// key : dato
```

```
// next : puntatore al nodo successivo
// prev : puntatore al nodo precedente
```

Nota. La lista doppiamente concatenata facilita la navigazione in entrambe le direzioni e semplifica alcune operazioni (come la cancellazione).

6.0.8 Operazioni su liste doppiamente concatenate

6.0.8.1 Ricerca

```
Node listSearch(List L, int k){
    Node x = L.head;
    while((x != NIL) && (x.key != k)){
        x := x.next;
    }
    return x;
}
```

Complessità: $O(n)$

6.0.8.2 Inserimento in testa

```
listInsert(List L, Node x){
    x.next := L.head;
    if(L.head != NIL){
        L.head.prev := x;
    }
    L.head := x;
    x.prev := NIL;
}
```

Complessità: $O(1)$

6.0.8.3 Cancellazione

```
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;
    }
}
```

Complessità: $O(1)$ (se abbiamo già il puntatore a x)

Nota. Se dobbiamo prima cercare l'elemento, la cancellazione diventa $O(n)$.

6.0.9 Varianti

- **Lista circolare:** l'ultimo elemento punta al primo
- **Lista con sentinella:** un nodo fittizio semplifica la gestione dei casi limite (lista vuota, inserimento in testa/coda)

6.0.10 Confronto Array vs Liste

Operazione	Array	Lista
Accesso per indice	$O(1)$	$O(n)$
Ricerca	$O(n)$	$O(n)$
Inserimento in testa	$O(n)$	$O(1)$
Inserimento in coda	$O(1)^*$	$O(1)^{**}$
Cancellazione	$O(n)$	$O(1)^{***}$

Tabella 19: * se c'è spazio, ** se manteniamo un puntatore alla coda, *** se abbiamo il puntatore al nodo

6.0.11 Pile (Stack)

Definizione 3 (Pila). Una **pila** (stack) è una struttura dati con politica **LIFO** (Last In, First Out): l'ultimo elemento inserito è il primo ad essere rimosso.

6.0.12 Operazioni

- **Push(S, x):** inserisce x in cima alla pila
- **Pop(S):** rimuove e restituisce l'elemento in cima
- **Top(S)** o **Peek(S):** restituisce l'elemento in cima senza rimuoverlo
- **IsEmpty(S):** verifica se la pila è vuota

6.0.13 Implementazione su Array

// Pila implementata con array A[1..n]
 // top = indice dell'elemento in cima (0 se vuota)

```
bool stackEmpty(Stack S){
    return S.top == 0;
}

push(Stack S, int x){
    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];
}
```

6.0.14 Implementazione su Lista

// Pila implementata con lista concatenata
 // head = puntatore all'elemento in cima

```
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;
}

```

6.0.15 Complessità

Tutte le operazioni hanno complessità $O(1)$.

6.0.16 Applicazioni

- Gestione delle chiamate di funzione (call stack)
- Valutazione di espressioni (notazione polacca inversa)
- Algoritmi di backtracking
- Undo/Redo in editor
- Bilanciamento delle parentesi

Esempio 1 (Bilanciamento parentesi). Per verificare se una stringa di parentesi è bilanciata:

```

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);
}

```

6.0.17 Code (Queue)

Definizione 4 (Coda). Una **coda** (queue) è una struttura dati con politica **FIFO** (First In, First Out): il primo elemento inserito è il primo ad essere rimosso.

6.0.18 Operazioni

- Enqueue(Q, x): inserisce x alla fine della coda
- Dequeue(Q): rimuove e restituisce l'elemento all'inizio

- `Front(Q)`: restituisce l'elemento all'inizio senza rimuoverlo
- `IsEmpty(Q)`: verifica se la coda è vuota

6.0.19 Implementazione su Array circolare

Per evitare lo spostamento degli elementi, usiamo un array «circolare»:

```
// Coda implementata con array Q[1..n]
// head = indice del primo elemento
// tail = indice della prossima posizione libera
```

```
enqueue(Queue Q, int x){
    Q[Q.tail] := x;
    if(Q.tail == Q.length){
        Q.tail := 1;
    } else {
        Q.tail := Q.tail + 1;
    }
}

int dequeue(Queue Q){
    int x = Q[Q.head];
    if(Q.head == Q.length){
        Q.head := 1;
    } else {
        Q.head := Q.head + 1;
    }
    return x;
}
```

Nota. Con l'array circolare, quando arriviamo alla fine dell'array, «torniamo all'inizio». Bisogna gestire i casi di overflow (coda piena) e underflow (coda vuota).

6.0.20 Implementazione su Lista

```
// Coda implementata con lista concatenata
// head = puntatore al primo elemento
// tail = puntatore all'ultimo elemento
```

```
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;
}

```

6.0.21 Complessità

Tutte le operazioni hanno complessità $O(1)$.

6.0.22 Applicazioni

- Scheduling di processi (CPU scheduling)
- Buffer di I/O
- Gestione delle richieste (web server)
- BFS (Breadth-First Search) nei grafi
- Simulazione di sistemi con attese

6.0.23 Varianti delle Code

6.0.24 Deque (Double-Ended Queue)

Definizione 5 (Deque). Una **deque** permette inserimenti e rimozioni da entrambe le estremità:

- InsertFront(D, x), InsertBack(D, x)
- RemoveFront(D), RemoveBack(D)

6.0.25 Coda con priorità

Già vista nel capitolo sugli Heap: gli elementi escono in base alla loro priorità, non all'ordine di arrivo.

6.0.26 Riepilogo

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	$O(1)$	$O(1)$	$O(1)^{**}$

Tabella 20: * in testa/coda con puntatore, ** solo all'elemento accessibile

Nota. La scelta della struttura dati dipende dalle operazioni più frequenti:

- Accesso casuale frequente → **Array**
- Inserimenti/cancellazioni frequenti → **Lista**
- Elaborazione LIFO → **Pila**
- Elaborazione FIFO → **Coda**