1.Introduzione

Contatti e informazioni generali

Dott. Simone Staccone
simone.staccone@uniroma2.it

Dipartimento di Ingegneria civile e Ingegneria informatica (DICII)
DAMON Research Group

20 Ottobre 2025

Informazioni Generali

Contatti:

simone.staccone@uniroma2.it

Microsoft Teams

Personal Website

Lezioni: (Da confermare)

- **Giorno:** Lunedì
- **Orario:** 16:00-18:00
- **Luogo:** Aula 3

Obiettivi del tutoraggio

- Saper stimare la complessità computazionale asintotica di algoritmi di base, avendo gli strumenti per formalizzare la stima di ricorrenze lineari.
- Comprendere i principi base della programmazione in C, avendo una visione sulla gestione della memoria e su come rappresentare le principali strutture dati.
- Sviluppare una capacità logica di base nella comprensione dei principali algoritmi e delle strutture dati di base.
- Svolgere esercitazioni e chiarire concetti in modo da poter superare l'esame nel migliore dei modi possibili!

Argomenti trattati

Analisi della complessità

- Introduzione all'analisi della complessità
- Metodologie per la soluzione di ricorrenze lineari
- Esercizi sulle ricorrenze lineari

Programmazione in C

- Processo di compilazione e comprensione del *memory layout* di un programma C.
- Gestione degli header file e dell'automazione della compilazione.
- Gestione della memoria statica e dinamica.
- Esercitazioni pratiche in C su problemi di logica di base.

Algoritmi di ordinamento

- BubbleSort, InsertionSort, MergeSort e QuickSort

Strutture dati di base

- Stack,Queue e Alberi
- Grafi e Hash Table

Algoritmi su grafi

- BFS e DFS
- Priority Queue e algoritmo di Dijkstra

Preparazione all'esame

- Esercizi vari
- Simulazioni d'esame
- Correzione di appelli d'esame svolti

Per analizzare la complessità di un algoritmo è necessario considerare due aspetti fondamentali:

- **Complessità spaziale:** quanta memoria occupa l'algoritmo.
- **Complessità temporale:** quanto tempo impiega a terminare.

Nel corso ci concentreremo principalmente sulla **complessità temporale**, ma i concetti trattati si applicano in modo analogo anche alla **complessità spaziale**.

Per poter studiare gli algoritmi in modo generale, abbiamo bisogno di un **modello astratto di calcolo**, che ci permetta di analizzarli:

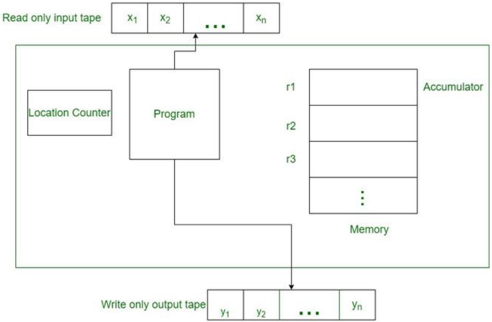
- indipendentemente dal linguaggio di programmazione;
- senza legarci alla macchina fisica o virtuale su cui vengono eseguiti;
- in termini puramente teorici, ma con implicazioni pratiche.

Come modello considereremo una **Macchina di Turing** di tipo **RAM (Random Access Memory)**.

In altre parole, useremo come astrazione un calcolatore con:

- memoria **infinita**;
- accesso **casuale** a ogni cella (come in un computer moderno);
- capacità di muoversi in **entrambi i sensi**.

Questo modello ci consente di analizzare il **tempo** e lo **spazio** richiesti dagli algoritmi, portando alla teoria dell'**analisi della complessità**.



Random Access Memory Model

Approfondimento: Turing Machine on GeeksforGeeks

C

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

C++

```
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std
        ::endl;
    return 0;
}
```

Java

```
public class Main {
    public static void main(String[]
         args) {
        System.out.println("Hello,
            World!");
    }
}
```

Python

```
print("Hello, World!")
```

JavaScript

```
console.log("Hello, World!");
```

- Per analizzare gli algoritmi, dobbiamo utilizzare un linguaggio astratto, che non tenga conto delle implementazioni pratiche e che abbia una corrispondenza univoca con qualsiasi altro linguaggio.
- Utilizzeremo quindi uno **pseudocodice**: un linguaggio comune per esprimere le operazioni di base di ogni linguaggio di programmazione (if, else, for, while, ecc.), considerando ogni operazione come avente un costo unitario in lettura e in scrittura.
- Non esiste uno standard universale per lo pseudocodice; l'importante è che ogni operazione rappresenti un'unità per poter effettuare stime quantitative ma qualitative del codice.
- Per provare online il pseudocodice, puoi usare il compilatore disponibile su: **Pseudocode Online Compiler**.

Complessità Computazionale

Pseudocodice

- **Tipi di dati:** integer, float, string, boolean
- **Assegnamento:** $x \leftarrow 5$
- **Restituzione:** `return` x
- **Controlli di flusso:** If, Else, For, While
- **Commenti:** %, // oppure $\triangleright$

Algorithm Somma di due vettori

```
function VECTORADD(integer[] A, integer[] B) : integer[]
    if length(A) ≠ length(B) then
        return
    end if
    C ← new integer[length(A)]
    for i = 1 to length(A) do
        C[i] ← A[i] + B[i]
    end for
    return C
end function
```

Complessità Computazionale

Pseudocodice (Somma di due numeri)

Calcolare quante righe di codice vengono eseguite dal seguente algoritmo

Algorithm Somma di due numeri

```
function SOMMA : integer
    read(a)
    read(b)
    sum ← a + b
    return sum
end function
```

- **Linee 1-2:** `read(a)`, `read(b)` $\Rightarrow$ 2 operazioni (lettura)
- **Linea 3:** `sum = a + b` $\Rightarrow$ 1 operazione (somma + assegnamento)
- **Linea 4:** `return sum` $\Rightarrow$ 1 operazione (resto valore)

Totale operazioni eseguite

Totale = 2 (letture) + 1 (somma) + 1 (return) = **4 operazioni**

Complessità Computazionale

Pseudocodice (Calcolo dei divisori di 100)

Calcolare quante righe di codice vengono eseguite dal seguente algoritmo

Algorithm Divisori di 100

```
function DIVISORI : integer
    i ← 0
    while i < 100 do
        if 100 mod i == 0 then
            print(i)
        end if
        i ← i + 1
    end while
end function
```

- **Linea 1:** `i ← 0` $\Rightarrow$ 1 operazione (assegnamento)
- **Linea 2:** `while (i < 100)` $\Rightarrow$ la condizione viene verificata 101 volte (100 volte vere + 1 volta falsa)
- **Linea 3:** `100 mod i == 0` $\Rightarrow$ 1 operazione per ogni iterazione valida (100 volte)
- **Linea 4:** `print(i)` $\Rightarrow$ eseguita solo quando la condizione è vera (i divisori di 100)
- **Linea 5:** `i ← i + 1` $\Rightarrow$ 1 operazione per ogni iterazione (100 volte)

Totale operazioni approssimative

1 (inizializzazione) + 101 (condizioni) + 100 (mod) + 100 (incrementi) + d (stampe)

dove d è il numero dei divisori di 100 ($d = 9$)

$\Rightarrow$ Totale $\approx 1 + 101 + 100 + 100 + 9 =$ **311 operazioni**

Calcolare quante righe di codice vengono eseguite dal seguente algoritmo

Algorithm Divisori di N

```
function DIVISORI(integer N)
    i ← 1
    while i ≤ N do
        if N mod i == 0 then
            print(i)
        end if
        i ← i + 1
    end while
end function
```

- **Linea 1:** `i ← 1` $\Rightarrow$ 1 operazione (assegnamento)
- **Linea 2:** `while (i` $\leq$ `N)` $\Rightarrow$ la condizione viene verificata $N+1$ volte (N volte vere + 1 volta falsa)
- **Linea 3:** `N mod i == 0` $\Rightarrow$ 1 operazione per ogni iterazione (N volte)
- **Linea 4:** `print(i)` $\Rightarrow$ eseguita solo quando la condizione è vera (dove d è il numero dei divisori di N)
- **Linea 5:** `i ← i + 1` $\Rightarrow$ 1 operazione per ogni iterazione (N volte)

Totale operazioni approssimative

1 (inizializzazione) + (N + 1) (condizioni) + N (mod) + N (incrementi) + d (stampe)

dove d è il numero dei divisori di N

$\Rightarrow$ Totale $\approx 1 + (N+1) + N + N + d = 3N + 2 + d$ operazioni

Calcolare quante righe di codice vengono eseguite dal seguente algoritmo

Algorithm Massimo Comun Divisore

```
function GCD(integer a, integer b) : integer
    while b ≠ 0 do
        x ← b
        b ← a mod b
        a ← x
    end while
    return a
end function
```

Algorithm Minimo Comune Multiplo

```
function MCM(integer a, integer b)
    c ← (a · b) / gcd(a, b)
    output c
end function
```

Funzione `gcd(a,b)`

- **Linea 1:** `while` $(b \neq 0) \to$ eseguita $k+1$ volte
- **Linee 2–4:** 3 operazioni per iterazione
- **Linea 5:** `return a` $\to 1$ operazione

Funzione `mcm(a,b)`

- **Linea 1:** `c ← (a*b)/gcd(a,b)` $\to 1$ moltiplicazione + 1 divisione + 1 chiamata a gcd
- **Linea 2:** `output c` $\to 1$ operazione

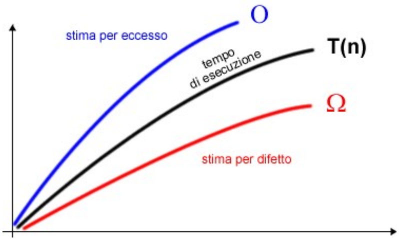
Totale operazioni (approssimativo)

gcd(a,b): $3k+2$ operazioni
mcm(a,b): 1 (moltiplicazione) $+\ 1$ (divisione) $+\ 3k+2$ (gcd) $+\ 1$ (output)
$\Rightarrow$ **Totale** $\approx 3k+5$ operazioni

Osservazioni

- Il numero di iterazioni k dipende dal rapporto tra a e b.
- L'algoritmo di Euclide è molto efficiente: $k \approx \log \min(a,b)$.

- La velocità di un algoritmo dipende dalla **dimensione dell'input**.
- L'analisi deve sempre essere **contestualizzata** rispetto al tipo di input considerato.
- Alcuni algoritmi risultano molto **efficienti per input piccoli**, ma diventano disastrosi su input grandi.
- Altri, come il `quicksort`, possono avere **comportamenti diversi** con input di pari dimensione.



"Si considera di solito il caso pessimo per stimare la complessità."

- $O(f(n))$: limite superiore asintotico.

$$T(n) = O(f(n)) \quad \text{se } \exists c > 0, n_0 \text{ t.c. } T(n) \leq cf(n), \forall n \geq n_0$$

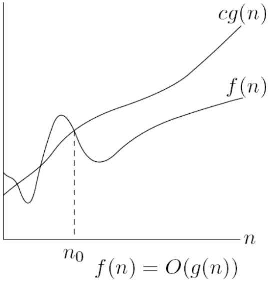
- $\Omega(f(n))$: limite inferiore asintotico.

$$T(n) = \Omega(f(n)) \quad \text{se } \exists c > 0, n_0 \text{ t.c. } T(n) \geq cf(n), \forall n \geq n_0$$

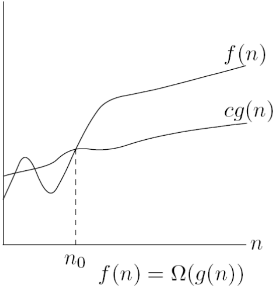
- $\Theta(f(n))$: limite stretto (O + Ω).

$$T(n) = \Theta(f(n)) \quad \text{se } T(n) = O(f(n)) \text{ e } T(n) = \Omega(f(n))$$

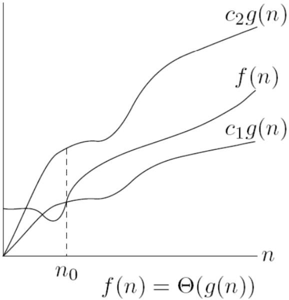
Esempi visivi delle notazioni asintotiche



O(f(n))



$\Omega(f(n))$



$\Theta(f(n))$