# Indice

# Capitolo 1

# Esercizi Complessità Asintotiche

**Esercizio 1.1.** *Ordinare in ordine crescente di grandezza le seguente funzioni giustificandone l'ordine:*
$n^{\log_2 \log_4^8 16}$ ; $n^{\log_2 \log_4 16^8}$ ; $64^{\log_{16} n}$

**Soluzione**

> Dobbiamo cercare di arrivare ad avere le funzioni nella forma più simile possibile per così poi facilmente poterle confrontare
>
> $n^{\log_2 \log_4^8 16} = n^{\log_2 (\log_4 16)^8} = n^{8 \log_2 2} = n^8$
> $n^{\log_2 \log_4 16^8} = n^{\log_2 8 \log_4 16} = n^{\log_2 16} = n^4$
> $64^{\log_{16} n} = 2^{6 \frac{\log_2 n}{\log_2 16}} = n^{\frac{6}{4}} = n^{\frac{3}{2}}$
>
> È facile vedere ora che l'ordinamento sarà: $n^8 > n^4 > n^{\frac{3}{2}}$

**Esercizio 1.2.** *Ordinare in ordine crescente di grandezza le seguente funzioni giustificandone l'ordine:*
$2^{\log_2 \sqrt{n}}$ ; $2^{n/2}$ ; $2^{\log_4 n}$ ; $8^{\log_4 n}$

**Soluzione**

> Dobbiamo cercare di arrivare ad avere le funzioni nella forma più simile possibile per così poi facilmente poterle confrontare
>
> $2^{\log_2 \sqrt{n}} = 2^{\frac{1}{2} \log_2 n} = n^{\frac{1}{2} \log_2 2} = \sqrt{n}$
> $2^{n/2}$, in questo caso non possiamo semplificare ulteriormente
> $2^{\log_4 n} = 2^{\frac{\log_2 n}{\log_2 4}} = 2^{\frac{1}{2} \log_2 n} = \sqrt{n}$
> $8^{\log_4 n} = 2^{3 \log_4 n} = 2^{3 \frac{\log_2 n}{\log_2 4}} = n^{\frac{3}{2}}$
>
> È facile vedere ora che l'ordinamento sarà: $2^{\frac{n}{2}} > n^{\frac{3}{2}} > \sqrt{n}$

**Esercizio 1.3.** *Ordinare in ordine crescente di grandezza le seguente funzioni giustificandone l'ordine:*
$n^{1/2}$ ; $4^{\log_{16} n}$ ; $3^{4 \log_3 n}$

**Soluzione**

Dobbiamo cercare di arrivare ad avere le funzioni nella forma più simile possibile per così poi facilmente poterle confrontare

Anche in questo caso $n^{1/2}$ non può essere ulteriormente semplificato
$4^{\log_{16} n} = 2^{2\frac{\log_2 n}{\log_2 16}} = n^{\frac{1}{2}\log_2 2} = \sqrt{n}$
$3^{4\log_3 n} = n^{4\log_3 3} = n^4$

È facile vedere ora che l'ordinamento sarà: $n^4 > n^{\frac{1}{2}}$

**Esercizio 1.4.** *Ordinare in ordine crescente di grandezza le seguente funzioni giustificandone l'ordine:*
$\log_2^6 n; n^{\frac{1}{3}}; 2^{2\log_2 \log_4 n}$

**Soluzione**

Dobbiamo cercare di arrivare ad avere le funzioni nella forma più simile possibile per così poi facilmente poterle confrontare

$\log_2^6 n = (\log_2 n)^6$
Anche in questo caso $n^{\frac{1}{3}}$ non può essere ulteriormente semplificato
$2^{2\log_2 \log_4 n} = 2^{2\log_2 \frac{\log_2 n}{\log_2 4}} = n^{2\log_2 \frac{1}{2}} = n^{-2}$

È facile vedere ora che l'ordinamento sarà: $\log_2^6 n > n^{\frac{1}{3}} > n^{-2}$

**Esercizio 1.5.** *Ordinare le seguenti funzioni:*
$n^{\frac{1}{\log_2 n}}; \log(n!); 2^{\sqrt{2\log_2 n}}$

**Soluzione**

Dobbiamo cercare di arrivare ad avere le funzioni nella forma più simile possibile per così poi facilmente poterle confrontare

$n^{\frac{1}{\log_2 n}} = n^{\frac{\log_2 2}{\log_2 n}} = 2^{\frac{\log_2 n}{\log_2 n}} = 2$
$\log(n!)$ per semplificare questa funzione ci interroghiamo sul significato di $n!$
$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \leq \underbrace{n \cdot n \cdot \ldots \cdot n}_{n \text{ volte}} = n^n$

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 \geq \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2}}_{\frac{n}{2} \text{ volte}} = \frac{n}{2}^{\frac{n}{2}}$$

Pertanto vale che $\frac{n}{2}^{\frac{n}{2}} \leq n! \leq n^n \implies \frac{n}{2}\log\frac{n}{2} \leq \log(n!) \leq n\log n \implies \in \Theta(n\log n)$
$2^{\sqrt{2\log n}} = 2^{(2\log_2)^{\frac{1}{2}}} = n^{(2\log_2 2)^{\frac{1}{2}}} = n^{\sqrt{n}}$

È facile vedere ora che l'ordinamento sarà: $n^{\sqrt{n}} > n\log n > 2$

**Esercizio 1.6.** *Applicando la definizione di ordine di grandezza, provare o confutare che:*

1. $f_1(n) = 2^{3n+3}, f_1(n) \in \Theta(2^n)$
2. $f_2(n) = 2^{n+5}, f_2(n) \in \Theta(2^n)$
3. $f_3(n) = n^2 + 7n + \log n, f_3(n) \in O(n^2)$
4. $f_4(n) = 2n + \log n, f_4(n) \in O(n)$

**Soluzione**

Al fine di risolvere tale tipologia di esercizio è necessario applicare la definizione di ordine di grandezza, e risolvere le disequazioni al fine di trovare i valori di $c$ e $n_0$ tali per cui la definizione vale (o verificare che non esiste alcun valore per cui la definizione è valida)
**(1):** Partiamo con lo scrivere la definizione di $\Theta(\cdot)$
**NB:** ricordiamoci che $\Theta(\cdot) \implies \mathcal{O}(\cdot) \wedge \Omega(\cdot)$, quindi ciò significa che devono valore contemporaneamente $\mathcal{O}(\cdot)$ e $\Omega(\cdot)$.
$\exists c_1 > 0, c_2 > 0$ e $n_0 \geq 1$ t.c. $\forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

- $2^{3n+3} \in \Omega(2^n)$
  $2^{3n+3} \geq c_1 2^n$
  $2^n \cdot 2^n \cdot 2^{n+3} \geq c_1 2^n$
  $2^n \cdot 2^{n+3} \geq c_1 \implies$ questo è sempre vero già con $n_0 = 1$ e $c_1 = 1 \implies 2^{3n+3} \in \Omega(2^n)$
- $2^{3n+3} \in \mathcal{O}(2^n)$
  $2^{3n+3} \leq c_2 2^n$
  $2^n \cdot 2^n \cdot 2^{n+3} \leq c_2 2^n$
  $2^n \cdot 2^{n+3} \leq c_2$
  $\implies \nexists$ costante $c_2$ maggiore di una funzione monotona crescente $\implies 2^{3n+3} \notin \Omega(2^n)$

Pertanto, $2^{3n+3} \notin \Theta(2^n)$.

Verifichiamo se $f_2(n) = 2^{n+5}, f_2(n) \in \Theta(2^n)$.
**(2):** $\exists c_1 > 0, c_2 > 0$ e $n_0 \geq 1$ t.c. $\forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

- $2^{n+5} \in \Omega(2^n)$
  $2^n \cdot 2^5 \geq c_1 \cdot 2^n$
  $c_1 \leq 2^5 \implies$ per $c_1 = 2^5$ e ad esempio $n_0 = 1$, $2^{n+5} \in \Omega(2^n)$
- $2^{n+5} \in \mathcal{O}(2^n)$
  $2^{n+5} \leq c_2 \cdot 2^n$
  $2^n \cdot 2^5 \leq c_2 \cdot 2^n$
  $c_2 \geq 2^5 \implies$ per $c_2 = 2^5$ e ad esempio $n_0 = 1$, $2^{n+5} \in \mathcal{O}(2^n)$

Verifichiamo se $f_3(n) = n^2 + 7n + \log n, f_3(n) \in O(n^2)$.
**(3):** $\exists c_1 > 0,$ e $n_0 \geq 1$ t.c. $\forall n \geq n_0, f(n) \leq c_1 g(n)$

- $n^2 + 7n + \log n \in \mathcal{O}(n^2)$
  $n^2 + 7n + \log n \leq c_2 \cdot n^2$
  Osserviamo che $\log_2 n \leq n$ già per un valore di $n_0 = 1$

$n^2 + 8n \leq c_2 \cdot n^2$
Osserviamo che $8n \leq 8n^2$ già per un valore di $n_0 = 1$
$9n^2 \leq c_2 \cdot n^2 \implies c_2 \geq 9$
Quindi, con $c_2 = 9$ e $n_0 = 1$, $n^2 + 7n + \log n \in \mathcal{O}(n^2)$.

Verifichiamo se $f_4(n) = 2n + \log n, f_4(n) \in O(n)$
**(4):** $\exists c_1 > 0,$ e $n_0 \geq 1$ t.c. $\forall n \geq n_0, f(n) \leq c_1 g(n)$
$2n + \log n, f_4(n) \leq c_1 \cdot n$
Come fatto in precedenza maggioriamo il termine $\log_2 n \leq n$, ed è vero già per $n_0 = 1$.
$3n \leq c_1 \cdot n$
$c_1 \geq 3 \implies$ per $n_0 = 1$ e $c_1 = 3, 2n + \log n, f_4(n) \in O(n)$.

**Esercizio 1.7.** *Provare o confutare applicando la definizione di ordine di grandezza la seguente affermazione:*
$n^3 + \frac{1}{7}n^4 + 12 \log_2 n \in \Theta(n^4)$
**Soluzione**

Dobbiamo provare che $n^3 + \frac{1}{7}n^4 + 12 \log_2 n \in \Theta(n^4)$
Quindi, $\exists c_1 > 0, c_2 > 0$ e $n_0 \geq 1$ t.c. $\forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

- $n^3 + \frac{1}{7}n^4 + 12 \log_2 n \in \Omega(n^4)$
  Per provarlo, possiamo **minorare** la nostra funzione per semplificarci la vita ed eliminare i termini in "eccesso".
  In particolare, possiamo dire che $n^3 + \frac{1}{7}n^4 + 12 \log_2 n \geq \frac{1}{7}n^4$, ed è vero già per $n_0 = 1$
  $\frac{1}{7}n^4 \geq c_1 \cdot n^4$
  Quindi, osserviamo che l'appartenenza a $\Omega(n^4)$ è verificata per $n_0 = 1$ e ad esempio $c = \frac{1}{7} \implies n^3 + \frac{1}{7}n^4 + 12 \log_2 n \in \Omega(n^4)$
- $n^3 + \frac{1}{7}n^4 + 12 \log_2 n \in \mathcal{O}(n^4)$
  Con la stessa logica precedente, ora, possiamo **maggiorare**: $12 \log_2 n \leq 12n^3$ ed è valido da $n_0 = 1$
  $\frac{1}{7}n^4 + 13n^3 \leq c_2 \cdot n^4$
  Ancora proviamo a maggiorare per rendere maggiormente evidente l'appartenenza: $13n^3 \leq 13n^4$, verificato da $n_0 = 1$
  $\frac{92}{7} \cdot n^4 \leq c_1 \cdot n^4 = c_1 \geq \frac{92}{7} \implies$ Per $c = \frac{92}{7}$ e $n_0 = 1$, $n^3 + \frac{1}{7}n^4 + 12 \log_2 n \in \mathcal{O}(n^4)$.

Abbiamo dunque provato che $n^3 + \frac{1}{7}n^4 + 12 \log_2 n \in \Theta(n^4)$.

**Esercizio 1.8.** *Provare o confutare applicando la definizione di ordine di grandezza la seguente affermazione:*

- $2^{n+4} \in \Theta(3^n)$
- $f(n) + g(n) \in \Theta(\min(f(n), g(n)))$

**Soluzione**

**Esercizio (1):** $2^{n+4} \in \Theta(3^n)$
$\exists c_1 > 0, c_2 > 0$ e $n_0 \geq 1$ t.c. $\forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

- $2^{n+4} \in \Omega(3^n)$
  $2^{n+4} \geq c_1 \cdot 3^n$
  $(\frac{2}{3})^n \cdot 2^4 \geq c_1$
  Dato che $n \Rightarrow +\infty$ il lato sinistro della disequazione tenderà a 0, mentre il lato destro
  $c_1 > 0 \implies 2^{n+4} \notin \Omega(3^n)$
  Segue che $2^{n+4} \notin \Theta(3^n)$

**Esercizio (2):** $f(n) + g(n) = \begin{cases} \mathcal{O}(\min(f(n), g(n))), & \text{and} \\ \Omega(\min(f(n), g(n))), \end{cases}$

Cominciamo con $\Omega$:

- Senza perdita di generalità assumiamo che $f(n)$ sia più piccola asintoticamente di $g(n)$. Abbiamo che $f(n) \geq \min(g(n), f(n))$, e $g(n) \geq \min(g(n), f(n))$ sono verificati per la definizione di min. Quindi da questo segue che $f(n) + g(n) \geq c \min(g(n), f(n))$; questo è vericato ad esempio per una qualunque costante $c$, e.g., $c = 2$
  $\implies f(n) + g(n) \in \Omega(\min(f(n), g(n)))$.

Rimane da verificare $\mathcal{O}$:

- Senza perdita di generalità assumiamo che $f(n) = n$ e che $g(n) = 1$, quindi:

  $$f(n) + g(n) \leq \min(g(n), f(n)) \Rightarrow n + 1 \leq \underbrace{\min(n, 1)}_{=\mathcal{O}(1)}$$

  Pertanto questa disequazione non è vera per nessuna costante $c > 0$, quindi $f(n) + g(n) \notin \mathcal{O}(\min(g(n), f(n)))$

Segue che $f(n) + g(n) \notin \Theta(\min(g(n), f(n)))$

**Esercizio 1.9.** *Per uno stesso problema sono stati pensati tre diversi algoritmi le cui complessità in tempo sono:*

- $f_1(n) = \sum_{k=1}^{n} (k^2 + k)$
- $f_2(n) = \sum_{k=0}^{\log_2 n} (\lceil \frac{n^2}{2^k} \rceil)$

- $f_3(n) = \sum_{k=1}^{n} \sum_{j=1}^{n} \Theta(1)$

**Soluzione**

Per decidere quale sia l'algoritmo più efficiente è neccesario studiare il valore di $f_1(n), f_2(n), f_3(n)$.

1. $\sum_{k=1}^{n}(k^2+k) = \sum_{k=1}^{n} k^2 + \sum_{k=1}^{n} k = \frac{n(n+1)(2n+1)}{2} + \frac{n(n+1)}{2} = \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$
2. $\sum_{k=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n^2}{2^k} \right\rceil = n^2 \sum_{k=0}^{\lfloor \log_2 n \rfloor} (\lceil \frac{1}{2} \rceil)^k < n^2 \sum_{k=0}^{\infty} (\lceil \frac{1}{2} \rceil)^k = 2n^2 = \Theta(n^2)$
3. $f_3(n) = \sum_{k=1}^{n} \sum_{j=1}^{n} \Theta(1) = \sum_{k=1}^{n} n = n^2 \in \Theta(n^2)$

Ora, si può facilmente osservare che l'algoritmo più performante è quello descritto dalla funzione costo $\Theta(n^2)$ (algoritmo con funzione costo $f_2$ e $f_3$).

# Capitolo 2

# Esercizi Porzioni di Codice Iterativo

**Esercizio 2.1.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    k = n
    while k > 1 do
    {
        k = k - 2
    }
end
```

**Soluzione**

Per capire la complessità temporale della funzione TEST è necessario capire quante iterazioni il ciclo WHILE svolge, e quanto è il costo computazionale delle operazioni al suo interno. Per farlo osserviamo innanzitutto che l'assegnazione al suo interno $k = k - 2$ la possiamo considerare come una operazione *primitiva*, pertanto con costo costante $\Theta(1)$. Ora rimane da capire quante volte il suddetto ciclo viene ripetuto. Per determinarlo dobbiamo identificare:

- **condizione di arresto:** $k > 1$; ci fermiamo quando il valore di $k \leq 1$
- **funzione crescita/decremento del contatore:** $k = k - 2$;

Per individuare la funzione che si cela dietro al passo di decremento della variabile $k$, ci agevoliamo con la segunte tabella:

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **k:** $n$ | $n-2$, | $n-4$, | $n-6$, | ... | $n-2i$ |

Conoscendo che al passo generico $i$ la funzione che muove la nostra $k = n - 2i$ possiamo calcolare il numero di passi che il nostro ciclo dovrà svolgere chiedendoci quando è che $n - 2i \leq 1$.
Quindi:
$n - 2i \leq 1$
$i \geq \frac{n-1}{2} \implies$ la funzione TEST $\in \Theta(n)$.

**Esercizio 2.2.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    k = 1
    while k < n do
    {
        k = k * 3
    }
end
```

**Soluzione**

Come fatto nell'esercizio precedente individuiamo:

- **condizione di arresto:** $k < n$; ci fermiamo quando il valore di $k \geq n$
- **funzione crescita/decremento del contatore:** $k = k * 3$;

Per individuare la funzione che si cela dietro al passo di incremento della variabile $k$, ci agevoliamo con la segunte tabella:

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **k:** 1 | 3, | 9, | 27, | ... | $3^i$ |

Conoscendo che al passo generico $i$ la funzione che muove la nostra $k = 3^i$ possiamo calcolare il numero di passi chiedendoci quando è che $3^i \geq n$.
Quindi:
$3^i \geq n$
$i \log_3 3 \geq \log_3 n \implies$ la funzione TEST $\in \Theta(\log n)$.

**Esercizio 2.3.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    k = 1
    while k < n^2 do
    {
        k = k * 2
    }
end
```

**Soluzione**

Come fatto nell'esercizio precedente definiamo:

- **condizione di arresto:** $k < n^2$; ci fermiamo quando il valore di $k \geq n^2$
- **funzione crescita/decremento del contatore:** $k = k * 2$;

Per individuare la funzione che si cela dietro al passo di incremento della variabile $k$, ci agevoliamo con la segunte tabella:

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **k:** 1 | 2, | 4, | 8, | ... | $2^i$ |

Conoscendo che al passo generico $i$ la funzione che muove la nostra $k = 2^i$ possiamo calcolare il numero di passi chiedendoci quando è che $2^i \geq n^2$.
Quindi:
$2^i \geq n^2$
$i \log_2 2 \geq 2 \log_2 n$
$i \geq 2 \log_2 n \implies$ la funzione TEST $\in \Theta(\log n)$.

**Esercizio 2.4.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    k = √n
    while k > 1 do
    {
        k = k - 2
    }
end
```

**Soluzione**

Partiamo subito con il costruire la tabella che descrive il passo del nostro contatore:

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **k:** $\sqrt{n}$ | $\sqrt{n} - 2$, | $\sqrt{n} - 4$, | $\sqrt{n} - 6$, | ... | $\sqrt{n} - 2i$ |

Conoscendo che al passo generico $i$ la funzione che muove la nostra $k = \sqrt{n} - 2i$ possiamo calcolare il numero di passi chiedendoci quando è che $\sqrt{n} - 2i \leq 1$.

$\sqrt{n} - 2i \leq 1$
$2i \geq \sqrt{n} - 1$
$i \geq \frac{\sqrt{n}-1}{2} \implies \Theta(\sqrt{n})$.

**Esercizio 2.5.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    k = n
    while k > 1 do
    {
        k = log2 k
    }
end
```

**Soluzione**

Partiamo subito con il costruire la tabella che descrive il passo del nostro contatore:

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **k:** $n$ | $\log_2 n$, | $\log_2 \log_2 n$, | $\log_2 \log_2 \log_2 n$, | ... | $\underbrace{\log_2 \dots \log_2 n}_{i \text{ volte}} = \log_2^{(i)} n$ |

Conoscendo che al passo generico $i$ la funzione che muove la nostra $k = \log_2^{(i)} n$ possiamo calcolare il numero di passi chiedendoci quando è che $\log_2^{(i)} n \leq 1$.
Tuttavia riconosciamo in $\log_2^{(i)} n \leq 1$ la definizione di logaritmo iterato.
Quindi sapendo che
$\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$
notiamo subito che $log^*n$ è esattamente quello che dovevamo risolvere
$\implies$ TEST $\in \Theta(\log^* n)$.

**Esercizio 2.6.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    k = 0
    while k < n do
    {
        k = 2^k
    }
end
```

**Soluzione**

Partiamo subito con il costruire la tabella che descrive il passo del nostro contatore:

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **k:** $2^0 = 1$ | $2^{2^{(2^0)}} = 2^2$, | $2^{2^{2^{(2^0)}}} = 2^4$, | $2^{2^{2^{2^{(2^0)}}}} = 2^{16}$, | ... | $\underbrace{2^{2^{2^{\dots^{(2^0)}}}}}_{i \text{ volte}}$ |

Pertanto ci chiediamo quando è che $2^{2^{2^{\dots^{(2^0)}}}} \geq n$. Tuttavia quanto scritto appare poco chiaro. Osserviamo però che la nostra operazione di esponenziazione è stata ripetuta $i$ volte.

Ciò significa che potremmo "semplificare" le $i$ esponenziazioni iterando lo stesso numero di volte il logartimo ($\log_2$). Otterremo dunque che:

$\underbrace{\log_2 \log_2 \dots \log_2}_{i \text{ volte}} n \leq \underbrace{\log_2 \log_2 \dots \log_2}_{i \text{ volte}} \underbrace{2^{2^{2^{\dots^{(2^0)}}}}}_{i \text{ volte}}$

$\underbrace{\log_2 \log_2 \dots \log_2}_{i \text{ volte}} n \leq 1$

$\log_2^{(i)} n \leq 1$

Ma questo altro non è che la definizione di $\log^* n$ (**Definizione riportata nell'esercizio precedente**). Quindi la complessità della funzione TEST$\in \Theta(\log^* n)$.

**Esercizio 2.7.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    int k = 1
    for (l = 1; l ≤ n-1; l++)
    {
        for (i = 1; i ≤ 2^(l-1); i++)
        {
            A[3^(l-1)+i] = k
        }
        k = k + 1
    }
end
```

**Soluzione**

Differentemente dai precedenti esercizi, in questo programma abbiamo due cicli FOR innestati. Questa struttura può essere studiata valutando la complessità del ciclo più interno, e poi iterandola per il numero di iterazioni del ciclo più esterno. Questo suggerisce la possibilità di scrivere i nostri cicli FOR come delle sommatorie.
Quindi:
$\sum_{l=1}^{n-1} \sum_{i=1}^{2^{l-1}} \Theta(1) = \sum_{l=1}^{n-1} 2^{l-1}$
Quanto abbiamo ottenuto assomiglia a una serie geometrica, quello che possiamo fare è **maggiorare**
$\leq \sum_{l=0}^{n} 2^{l-1} = \frac{2^{n+1}-1}{2-1} \implies \text{TEST} \in \Theta(2^n)$
Questa maggiorazione è possibile perchè la nuova sommatoria differisce dalla precedente solo per termini positivi

**Esercizio 2.8.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    int s = 0; a=1
    for (l = 1; l ≤ ⌊log2 n⌋; l++)
    {
        s = s + 1
        a = a * s
    }
    while s ≥ 1 do
    {
        s = ⌊s/3⌋
    }
end
```

**Soluzione**

Questo esercizio propone un programma in cui ci sono due cicli non innestati, ma i cui indici sono in relazione. In particolare, il primo ciclo FOR incrementa l'indice $s$ che governa il secondo ciclo WHILE. Pertanto, al fine di studiarne la complessità dobbiamo determinare il costo del primo ciclo, tenendo traccia del valore assunto da $s$, e successivamente definire il numero di iterazioni del secondo ciclo. Infine si ritornerà il massimo fra le due funzioni, i.e., la funzione che cresce più lentamente asintoticamente.

- **(primo ciclo):** $\sum_{l=1}^{\lfloor \log_2 n \rfloor} \Theta(1) = \lfloor \log_2 n \rfloor \leq \log_2 n \in \Theta(\log n)$
  $\implies s \leq \log_2 n$; questo perchè $s$ cresce con lo stesso passo di $l$.
- **(secondo ciclo):** costruiamo la nostra solita tabella per capire il passo

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **s:** $\log_2 n$ | $\frac{\log_2 n}{3}$, | $\frac{\log_2 n}{9}$, | $\frac{\log_2 n}{27}$, | ... | $\frac{\log_2 n}{3^i}$ |

Quindi:
$\frac{\log_2 n}{3^i} < 1$
$i \log_3 3 > \log_3 \log_2 n$
$i > \log_3 \log_2 n \in \Theta(\log \log n) \implies \text{TEST} \in \max\{\Theta(\log n), \Theta(\log \log n)\} = \Theta(\log n)$.

**Esercizio 2.9.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    int s = n; a = 1; t = 0
    while s ≥ 1 do
    {
        s = ⌊s/2⌋
        a++
    }
    for (i = 1; i ≤ 2^a; i++)
    {
        t = 2 + t
    }
end
```

**Soluzione**

Per stimare la complessità della funzione TEST dobbiamo calcolare il costo dei due cicli. Cominciamo con il primo, partendo dal descrivere la funzione passo con la tabella.

- **(primo ciclo):**

| **step:** 0 | 1, | 2, | 3, | ... | $j$ |
|---|---|---|---|---|---|
| **s:** $n$ | $\lfloor\frac{n}{2}\rfloor$, | $\lfloor\frac{n}{4}\rfloor$, | $\lfloor\frac{n}{8}\rfloor$, | ... | $\lfloor\frac{n}{2^j}\rfloor$ |

  Quindi sapendo che al passo generico $j$ la funzione decremento che muove $s$ è uguale a $\lfloor\frac{n}{2^j}\rfloor$, possiamo calcolare il numero di passi necessari al ciclo WHILE per convergere risolvendo la seguente disquazione.
  $\lfloor\frac{n}{2^j}\rfloor < 1$
  Maggioriamo $\lfloor\frac{n}{2^j}\rfloor \leq \frac{n}{2^j}$ e risolviamo
  $\frac{n}{2^j} < 1$
  $\frac{1}{2^j} < \frac{1}{n}$
  $2^j > n$
  $j \log_2 2 > \log_2 n \implies \Theta(\log n)$

- **(secondo ciclo):** A questo punto dobbiamo studiare il costo del secondo ciclo FOR. Tuttavia per farlo è necessario conoscere il valore di $a$. Dato che $a$ viene incrementata di 1 ad ogni passo del primo ciclo, e quest'ultimo viene ripetuto $\log_2 n$, avremo che $a = \log_2 n$.
  Pertanto il ciclo FOR ci costa: $\sum_{i=1}^{2^a} \Theta(1) = 2^a = 2^{\log_2 n} = n \implies \Theta(n)$.

Sapendo ora che il costo del primo ciclo è $\Theta(\log n)$ e del secondo è $\Theta(n)$, possiamo concludere che il costo totale della funzione è il massimo fra i due costi asintotici; quindi TEST $\in \Theta(n)$.

**Esercizio 2.10.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Test (n)
begin
    int s = 0;
    for(i = 1; i ≤ n; i++)
    {
        s = s + 3^i
    }
    while s > 3^(n/2) do
    {
        s = s/9
    }
end
```

**Soluzione**

Anche in questo caso abbiamo un due cicli, dove il primo incrementa l'indice del secondo.

- **(primo ciclo):**

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **s:** $3^0 = 1$ | $3^0 + 3^1$, | $3^0 + 3^1 + 3^2$, | $3^0 + 3^1 + 3^2 + 3^3$, | ... | $3^0 + 3^1 + 3^2 \dots 3^i$ |

  Ci suggerisce che il valore di $s = \sum_{i=1}^{n} 3^i$
  Quindi:
  $\sum_{i=1}^{n} 3^i \leq \sum_{i=0}^{n} 3^i = \frac{3^n+1-1}{3-1}$
  Abbiamo che il primo ciclo appartiene a $\Theta(n)$ e il valore di $s = \frac{3^n+1-1}{2}$

- **(secondo ciclo):** Costruiamo la nostra solita tabella per studiare il passo del ciclo

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **s:** $\frac{3^{n+1}-1}{2}$ | $\frac{3^{n+1}-1}{2} \cdot \frac{1}{9}$, | $\frac{3^{n+1}-1}{2} \cdot \frac{1}{9^2}$, | $\frac{3^{n+1}-1}{2} \cdot \frac{1}{9^3}$, | ... | $\frac{3^{n+1}-1}{2} \cdot \frac{1}{9^i}$ |

  Quindi:
  $\frac{3^{n+1}-1}{2} \cdot \frac{1}{9^i} \leq 3^{\frac{n}{2}}$
  $\frac{1}{9^i} \leq \frac{2}{3^{n+1}-1} \cdot 3^{\frac{n}{2}}$
  $9^i \geq \frac{3^{n+1}-1}{2} \cdot (\frac{1}{3})^{\frac{n}{2}}$
  $2i \log_3 3 \geq \log_3(\frac{3^{n+1}-1}{2} \cdot (\frac{1}{3})^{\frac{n}{2}})$
  Maggioriamo $\log_3(\frac{3^{n+1}-1}{2} \cdot (\frac{1}{3})^{\frac{n}{2}}) \leq \log_3(3^{n+1} \cdot (\frac{1}{3})^{\frac{n}{2}})$, dunque
  $2i \geq \log_3(3^{n+1} \cdot (\frac{1}{3})^{\frac{n}{2}})$
  $2i \geq (2n + 1 - \frac{n}{2}) \log_3 3$
  $i \geq \frac{n}{4} + \frac{1}{2} \implies \in \Theta(n)$
  Pertanto il programma ha un costo di $\Theta(n)$.

**Esercizio 2.11.** *Stimare la complessità temporale della seguente porzione di codice.*

```
Procedure Pippo (n)
begin
    if n ≤ 3 then
        return 1
    else
        int a = n, i = 0
        while (a > 2) AND (i < √n) do
        {
            a = a − 2
            i++
        }
end
```

**Soluzione**

L'esercizio proposto, differentemente da quelli visti, presenta una serie di operazioni innestati all'interno di un IF. Ricordandoci che la struttura di selezione valuta una condizione, e nel caso in cui questa è verificata esegue una certa porzione di codice, viceversa ne esegue un'altra. Basandoci sul comportamento della selezione intuiamo che il costo computazionale della funzione PIPPO sarà dato dal ramo dell'IF computazionalmente più pesante.

- **Ramo Vero:** La prima osservazione che facciamo riguarda il ramo vero, infatti la funzione entra nel ramo vero ogni qualvolta la dimensione di $n \leq 3$. Questo significa che a prescindere dalle operazioni che svolgeremo all'interno del ramo vero, la dimensione dell'input sarà al più 3. Pertanto per quanto complesse le operazioni da svolgere, avendo un limite superiore all'input, il costo delle operazioni sarà sempre delimitato da una costante ($\Theta(1)$).

- **Ramo Falso:** in questo caso la dimensione dell'input può essere arbitrariamente grande, quindi la porzione di codice di questo ramo determinerà il costo finale della funzione. Osserviamo che abbiamo un ciclo WHILE con due condizioni concatenate da AND logico, ciò significa che dovremo studiare quale delle due verrà invalidata prima. Per capirlo, studiamo la complessità di un WHILE con condizione di arresto, prima $a > 2$ e successivamente $i < \sqrt{n}$.

  Cominciamo con $a > 2$; il valore di $a$ parte da $n$, e all'interno del ciclo viene decrementata di 2 ad ogni passo. Costruiamoci la tabella:

| **step:** 0 | 1, | 2, | 3, | ... | $i$ |
|---|---|---|---|---|---|
| **a:** $n$ | $n-2$, | $n-4$, | $n-6$, | ... | $n-2i$ |

  Segue che:
  $n - 2i \leq 2$
  $i \geq \frac{n-2}{2} \implies \Theta(n)$

  Facciamo la stessa cosa con la seconda condizione ($i < \sqrt{n}$).

| **step:** 0 | 1, | 2, | 3, | ... | $j$ |
|---|---|---|---|---|---|
| **i:** 0 | 1, | 2, | 3, | ... | $j$ |

Segue che:
$j \geq \sqrt{n} \implies \Theta(\sqrt{n})$

Sappiamo ora determinare la complessità del ciclo WHILE, infatti questa è uguale a $\min\{\Theta(n), \Theta(\sqrt{n})\} = \Theta(\sqrt{n})$. Quindi la complessità di PIPPO$\in \Theta(\sqrt{n})$.