Università "Sapienza" di Roma
Facoltà di Informatica

Introduzione agli Algoritmi

Author
Simone Bianco

17 marzo 2022

Indice

Capitolo 0

Introduzione

Si può facilmente osservare che al giorno d'oggi l'informatica **permea la nostra vita quotidiana**, sia quando essa è direttamente percepibile, sia quando è invisibile.

L'informatica viene spesso erroneamente considerata una mera attività pratica, per svolgere la quale è sufficiente un approccio dilettantistico e per cui non è necessaria una vera professionalità. Nulla di più inesatto: in realtà l'informatica è una **disciplina scientifica**.

Essa non può essere considerata una sorta di "scienza dei calcolatori", poiché **i calcolatori** (o elaboratori) ne **sono solo uno strumento**: l'informatico può anche lavorare solamente con carta e penna. In realtà, l'informatica, intesa come disciplina scientifica, non coincide con alcuna delle sue applicazioni.

In questo corso verranno date le definizioni di **algoritmo, strutture dati, efficienza, problemi computazionali** ed **ottimizzazione** di essi. Viene inoltre fornito un **modello teorico di calcolatore** che consentirà di paragonare tra loro algoritmi diversi che risolvono lo stesso problema.

Capitolo 1

Algoritmi, Efficienza e RAM

1.1 Algoritmi e Strutture Dati

La definizione di informatica proposta dall'**ACM (Association for Computing Machinery)**, nonché una delle principali organizzazioni scientifiche di informatici di tutto il mondo, è la seguente: *"**L'informatica è la scienza degli algoritmi** che descrivono e trasformano l'informazione: la loro teoria, analisi, progetto, efficienza, realizzazione e applicazione."*

Gli **algoritmi**, dunque, sono un concetto fondamentale per l'informatica, fino ad esserne il fulcro. Ma cosa intendiamo per algoritmo?

Definition 1. Algoritmo

Un **algoritmo** è "**una sequenza di comandi elementari** ed univoci che terminano in un **tempo finito** ed operano su **strutture dati**".

Un comando viene definito **elementare** quando **non può essere scomposto** in comandi più semplici. I comandi elementari sono quindi **univoci** e possono essere interpretati in un solo modo.

Se un algoritmo è **ben specificato**, chi (o ciò che) lo esegue non ha bisogno di pensare, deve solo eseguire con precisione i passi elencati nell'algoritmo nella sequenza in cui appaiono. Se un calcolatore esegue un algoritmo e l'output è errato, **la colpa non è del calcolatore, ma del progettista**.

Prima di poter risolvere un problema abbiamo, ovviamente, bisogno di pensare ad un modo per poter **gestire i dati** che vengono utilizzati dall'algoritmo stesso. A tal fine, sarà necessario definire le opportune **strutture dati** su cui opererà l'algoritmo, ossia gli strumenti necessari per **organizzare** e **memorizzare** i dati veri e propri, semplificandone l'accesso e la modifica.

È importante sottolineare che **non esiste una struttura dati che sia adeguata per ogni problema**, dunque è necessario conoscere proprietà, vantaggi e svantaggi delle principali strutture dati in modo da poter scegliere di volta in volta quale sia quella **più adatta al problema**.

La scelta della struttura dati da adottare nella soluzione di un problema è un **aspetto fondamentale** per la risoluzione del problema stesso, al pari del progetto dell'algoritmo stesso. Per questa ragione, gli algoritmi e le strutture dati fondamentali vengono sempre studiati e illustrati assieme.

1.2 Efficienza di un algoritmo

Un aspetto fondamentale che va affrontato nello studio degli algoritmi è la loro **efficienza**, cioè la quantificazione delle loro **esigenze in termini di tempo e di spazio**, ossia tempo di esecuzione e quantità di memoria richiesta.

La scelta di un algoritmo rispetto ad altro, nel caso i due algoritmi portino allo stesso risultato, è molto importante:

- I calcolatori sono molto veloci, ma non infinitamente veloci
- La memoria è economica e abbondante, ma non è né gratuita né illimitata.

Un parametro fondamentale per la scelta dell'algoritmo è proprio la quantità di risorse spazio-tempo utilizzate. Nelle sezioni successive vedremo il concetto di **costo computazionale** degli algoritmi in termini di numero di operazioni elementari e quantità di spazio di memoria necessario in funzione della dimensione dell'input.

Esempio di valutazione dell'efficienza

Immaginiamo di voler risolvere il seguente problema: vogliamo **ordinare una lista di $n = \mathbf{10^6}$ numeri interi.** Vista l'enorme quantità di numeri, decidiamo di far svolgere questo compito ad un elaboratore. A nostra disposizione abbiamo **due calcolatori**:

- Un **calcolatore veloce**, che chiameremo **V**, in grado di svolgere 10^9 operazioni/sec
- Un **calcolatore lento**, che chiameremo **L**, in grado di svolgere 10^7 operazioni/sec

Immaginiamo di essere in grado di saper sviluppare solo **due algoritmi di ordinamento** (di cui per ora non vedremo il funzionamento, ma solo le specifiche temporali):

- L'algoritmo **Insertion Sort**, richiedente $2n^2$ operazioni **(più lento)**
- L'algoritmo **Merge Sort**, richiedente $50n \cdot log(n)$ operazioni **(più veloce)**

Ci chiediamo se la maggior velocità del calcolatore V sia in grado di **contro-bilanciare** la maggior lentezza dell'algoritmo IS. Proviamo quindi a calcolare il costo temporale di entrambe le scelte (**ATTENZIONE:** con log intendiamo il **logaritmo in base 2**):

$$V(IS) = \frac{2 \cdot (10^6)^2 \text{ operazioni}}{10^9 \text{ operazioni/sec}} = 2000 \text{ sec} \approx 33 \text{ min}$$

$$L(MS) = \frac{50 \cdot 10^6 \cdot log(10^6) \text{ operazioni}}{10^7 \text{ operazioni/sec}} \approx 100 \text{ sec} \approx 1.5 \text{ min}$$

Notiamo quindi che, nonostante la differenza di caratteristiche hardware, **la scelta dell'algoritmo è cruciale per l'efficienza**.

Per ricalcare maggiormente il concetto, proviamo ad aumentare l'input a $n = 10^7$

$$V(IS) = \frac{2 \cdot (10^7)^2 \text{ operazioni}}{10^9 \text{ operazioni/sec}} \approx 55.5 \text{ ore} \approx 2.3 \text{ giorni}$$

$$L(MS) = \frac{50 \cdot 10^7 \cdot log(10^7) \text{ operazioni}}{10^7 \text{ operazioni/sec}} \approx 19.5 \text{ min}$$

Aumentando l'input di un solo ordine di grandezza, la **differenza** in termini di costi temporali dei due algoritmi è **abissale**.

1.2.1 Random Access Machine (RAM)

Nell'esempio precedentemente visto, abbiamo considerato **due macchine diverse**, dove una era più performante dell'altra. Ciò non è un fattore da ignorare, poiché ovviamente le caratteristiche hardware dell'elaboratore **influiscono** direttamente sulle **performance dell'algoritmo**: se nell'esempio precedente calcolassimo $V(MS)$ con $n = 10^6$, il tempo impiegato dall'algoritmo sarebbe circa **1 secondo**, rispetto ai 1**00 secondi** impiegati da L(MS).

Per poter valutare la **vera efficienza** di un algoritmo, dunque, è necessario quantificare le risorse che esso richiede per la sua esecuzione senza che tale analisi sia **influenzata da una specifica tecnologia** che, inevitabilmente, col tempo **diviene obsoleta**. Dunque, è necessario valutare l'algoritmo come se venisse eseguito da una **macchina astratta** rispettante queste caratteristiche, ossia la **Random Access Machine** (anche chiamata **Modello RAM**).

Definition 2. Random Access Machine

La **Random Access Machine (RAM)** è una macchina astratta, la cui validità e potenza concettuale risiede nel fatto che **non diventa obsoleta** con il progredire della tecnologia.

Le caratteristiche del modello RAM sono:

- Un **singolo processore** che esegue le operazioni **sequenzialmente**
- Esistono **solo operazioni elementari** e l'esecuzione di ciascuna delle quali richiede per definizione un **tempo costante** (es.: operazioni aritmetiche, letture, scritture, salto condizionato, ecc.)
- Esiste un **limite alla dimensione** di ogni valore memorizzato ed al numero complessivo di valori utilizzati, dipendente dalle dimensioni delle word in memoria

1.2.2 Misura di Costo Uniforme

Sia d la **dimensione di bit di ogni parola** contenuta in memoria. Se è soddisfatta l'ipotesi che ogni dato in input sia un valore **minore** di 2^d, ciascuna operazione elementare sui dati del problema verrà eseguita in un **tempo costante**. In tal caso si parla di **misura di costo uniforme**.

Tale criterio **non è sempre realistico**: se un dato del problema non rispetta tale ipotesi, esso dovrà essere comunque memorizzato. In tal caso, sarà necessario usare **più parole di memoria** e, di conseguenza, anche le operazioni elementari su di esso dovranno essere reiterate per tutte le parole di memoria che lo contengono, richiedendo quindi un tempo non più costante. Per questo motivo, in ambito scientifico viene utilizzata la **misura di costo logaritmica**, più realistica rispetto a quella uniforme. Tuttavia, in questo corso essa **non verrà analizzata**.

Esempio di costo uniforme

Analizziamo il seguente codice:

```
def PotenzaDi2(n)
    x = 1
    for i in range(n):
        x = x*2
    return x
```

Il tempo di esecuzione totale è **proporzionale ad n**, poiché si tratta di un **ciclo eseguito** n **volte**, dove ad ogni iterazione vengono compiute tre operazioni, ciascuna di **costo unitario**:

- Viene incrementato il contatore relativo al ciclo for
- Viene calcolato $x \cdot 2$
- Viene assegnato il risultato del calcolo ad x

Capitolo 2

Notazione Asintotica

2.1 Notazione O grande, Omega e Teta

Come abbiamo accennato nel capitolo precedente, per poter **valutare l'efficienza** di un algoritmo, così da poterlo confrontare con algoritmi diversi che risolvono lo stesso problema, bisogna essere in grado di valutarne il suo **costo computazionale**, ovvero il suo tempo di esecuzione e/o le sue necessità in termini di memoria.

Questo tipo di valutazione, se effettuata nel dettaglio, risulta molto complessa e spesso contiene dettagli superflui. Per questo motivo, ci limiteremo a dare una visione più **astratta** e valutare solo quello che informalmente possiamo chiamare **tasso di crescita**, cioè la velocità con cui il tempo di esecuzione cresce all'aumentare della dimensione dell'input.

Tuttavia, poiché per piccole dimensioni dell'input il tempo impiegato è comunque poco indipendentemente dall'algoritmo, tale valutazione risulta più efficace quando la dimensione dell'input è **sufficientemente grande**. Per questo motivo, parleremo di **efficienza asintotica degli algoritmi**.

Definition 3. Notazione Asintotica

In matematica la **notazione asintotica** permette di confrontare il **tasso di crescita** (comportamento asintotico) di una funzione nei confronti di un'altra.

Il calcolo asintotico è utilizzato per analizzare la **complessità di un algoritmo**, stimandone in particolar modo, l'aumentare del **tempo di esecuzione** al crescere della **dimensione** n dell'input.

In particolare, esistono **tre tipologie di notazione asintotica**:

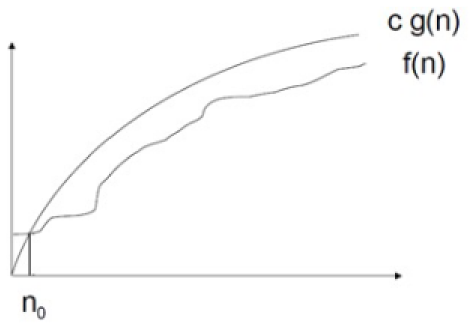
- **Notazione asintotica O grande**: definisce il limite superiore asintotico
- **Notazione asintotica** Ω: definisce il limite inferiore asintotico
- **Notazione asintotica** Θ: definisce il limite asintotico stretto

2.1.1 Notazione O grande

Per poter comprendere cosa si intende con **notazione O grande**, partiamo direttamente dalla sua definizione

Definition 4. O grande

Date due funzioni $f(n), g(n) \geq 0$ si dice che ***f(n)* è in *O(g(n))*** se esistono due costanti c ed n_0 tali che $0 \leq f(n) \leq c \cdot g(n)$ per ogni $n \geq n_0$



In $O(g(n))$, dunque, troviamo **tutte** le funzioni **«dominate»** dalla funzione $g(n)$

La notazione **O grande**, dunque, definisce quello che è il **limite superiore asintotico** della funzione $f(n)$: una volta superato un certo valore n_0 (dove $n \to +\infty$) l'andamento della funzione $f(n)$ viene **limitato** dalla funzione $c \cdot g(n)$, rimanendo sempre **al di sotto di essa** (dunque viene «dominata» da essa).

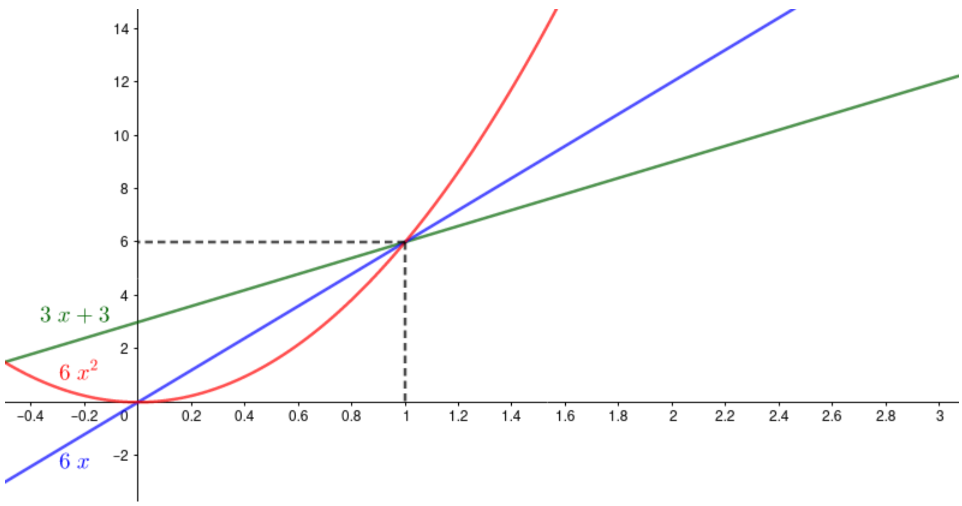
Esempi sull'O grande

- Sia $f(n) = 3n+3$. Possiamo dire che ***f(n)* è in *O(n²)***, in quanto esiste una almeno una c (ossia $c = 6$) per cui :

$$3n + 3 \leq c \cdot n^2, \quad \forall n \geq n_0, \quad c = 6, \quad n_0 = 1$$

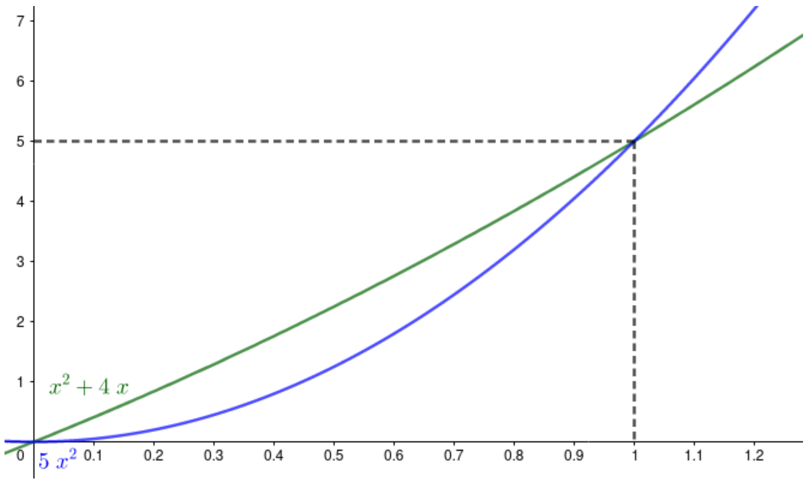
  Tuttavia, possiamo anche dire che **f(n) è in O(n)**, in quanto:

$$3n + 3 \leq c \cdot n, \quad \forall n \geq n_0, \quad c \geq 6, \quad n_0 = 1$$



- Sia $f(n) = n^2 + 4n$. Tale ***f(n)* è in *O(n²)*** in quanto:

$$n^2 + 4n \leq c \cdot n^2, \quad \forall n \geq n_0, \quad c \geq 5, \quad n_0 = 1$$



Notiamo come nel primo esempio abbiamo concluso che un **polinomio di primo grado** sia in $O(n)$, mentre nel secondo esempio abbiamo concluso che un **polinomio di secondo grado sia** in $O(n^2)$. Possiamo generalizzare la cosa nel seguente teorema:

Theorem 1

Sia $f(n)$ un **polinomio** di grado m, definito matematicamente come

$$f(n) = \sum_{i=0}^{m} a_i n^i = a_0 + a_1 n + a_2 n^2 + ... + a_m n^m$$

allora possiamo concludere che ***f(n)* è in *O(nᵐ)***

Dimostrazione per induzione

- **Caso base**: Abbiamo $m = 0$, per cui $f(n) = a_0 \cdot n^0$, dunque è una funzione costante e di conseguenza è in $O(1)$, che coincide con $O(n^0)$
- **Ipotesi induttiva**: Affermiamo che

$$\sum_{i=0}^{k} a_i n^i$$

è un $O(n^k)$ per ogni $k < m$, cioè esiste una costante c' tale che

$$\sum_{i=0}^{k} a_i n^i \leq c' \cdot n^k$$

- **Passo induttivo**: Dobbiamo dimostrare che

$$f(n) = \sum_{i=0}^{m} a_i n^i \leq c' \cdot n^m$$

Si osservi che, mettendo in evidenza l'ipotesi induttiva, $f(n)$ può essere riscritto come

$$f(n) = \sum_{i=0}^{m} a_i n^i = a_m n^m + \sum_{i=0}^{k} a_i n^i$$

per ogni $k < m$. Inoltre, per ipotesi induttiva, sappiamo che

$$\sum_{i=0}^{k} a_i n^i \leq c' \cdot n^k$$

dunque possiamo formulare la seguente catena di disuguaglianze

$$f(n) = a_m n^m + \sum_{i=0}^{k} a_i n^i \leq a_m n^m + c' \cdot n^k \leq a_m n^m + c' \cdot n^m$$

riscrivendo $a_m n^m + c' \cdot n^m$ come $(a_m + c') \cdot n^m$ e ponendo $c'' = a_m + c'$ otteniamo che

$$f(n) \leq c'' \cdot n^m$$

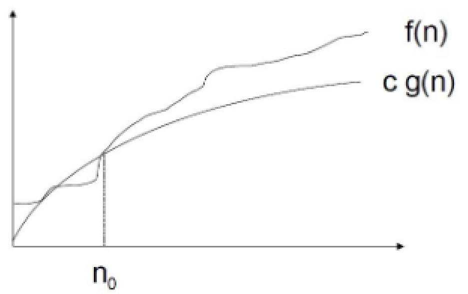
che per ipotesi sappiamo essere vera, dunque concludiamo che $f(n)$ è in $O(n^m)$

2.1.2 Notazione Omega

Nella sezione precedente, abbiamo definito la **Notazione O grande** come **limite superiore asintotico**. La **Notazione Omega**, invece, risulta essere il suo **opposto**:

Definition 5. Omega

Date due funzioni $f(n), g(n) \geq 0$ si dice che ***f(n)* è in Ω*(g(n))*** se esistono due costanti c ed n_0 tali che $f(n) \geq c \cdot g(n)$ per ogni $n \geq n_0$



In $O(g(n))$, dunque, troviamo **tutte** le funzioni che **«dominano»** dalla funzione $g(n)$

La notazione **Omega**, dunque, definisce quello che è il **limite inferiore asintotico** della funzione $f(n)$: una volta superato un certo valore n_0 (dove $n \to +\infty$) l'andamento della funzione $f(n)$ viene **limitato** dalla funzione $c \cdot g(n)$, rimanendo sempre **al di sopra di essa** (dunque «domina» essa).

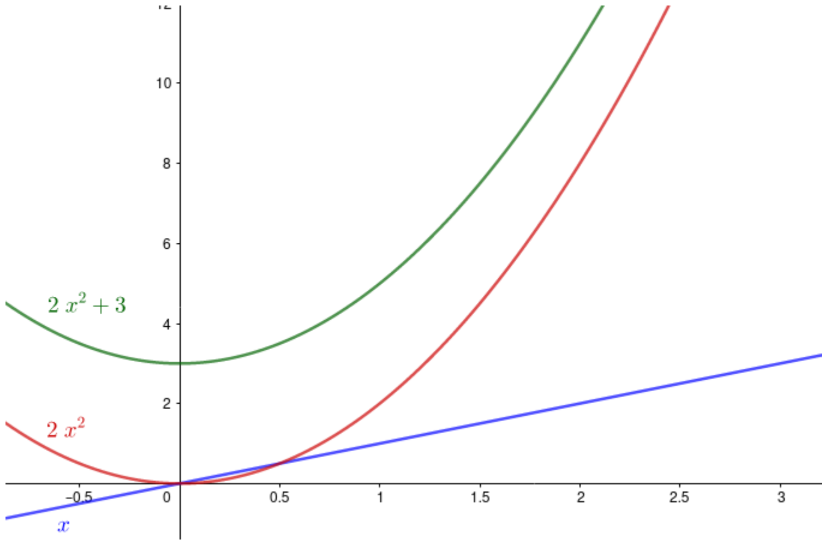
Esempi sull'Omega

- Sia $f(n) = 2n^2 + 3$. Possiamo dire che $f(n) = \Omega(n)$ in quanto

$$2n^2 + 3 \geq c \cdot n, \quad \forall n \geq n_0, \quad c = 1$$

  Tuttavia, possiamo anche dire che $f(n) = \Omega(n^2)$, in quanto:

$$2n^2 + 3 \geq c \cdot n^2, \quad \forall n \geq n_0, \quad c \leq 2$$



Analogamente alla **notazione O grande**, possiamo formulare il seguente teorema, la cui dimostrazione è analoga a quella già mostrata per O grande:

Theorem 2

Sia $f(n)$ un **polinomio** di grado m, definito matematicamente come

$$f(n) = \sum_{i=0}^{m} a_i n^i = a_0 + a_1 n + a_2 n^2 + ... + a_m n^m$$

allora possiamo concludere che ***f(n)* è in Ω*(n^m)***

2.1.3 Ordini di grandezza di O grande ed Omega

- Sia $f(n) = log(n)$. Allora $f(n)$ è in $O(\sqrt{n})$ e in $\Omega(1)$.

  Più in generale, abbiamo che:

  - $log^a(n) = O(\sqrt[b]{n})$ per ogni $a, b \geq 1$
  - $log^a(n) = \Omega(1)$ per ogni a
  - Dunque, possiamo dire che **un poli-logaritmo è dominato da qualunque radice** e che **un poli-logaritmo domina qualunque costante**

- Sia $f(n) = \sqrt[a]{n}$. Allora $f(n)$ è in $O(n)$ e in $\Omega(log(n))$.

  Più in generale, abbiamo che:

  - $\sqrt[a]{n} = O(n^b)$ per ogni $a, b \geq 1$
  - $\sqrt[a]{n} = \Omega(log^b(n))$ per ogni $a, b \geq 1$
  - Dunque, possiamo dire che **una radice è dominata da qualunque polinomio** e che **una radice domina qualunque poli-logaritmo**

- Sia $f(n) = n^a$. Allora $f(n)$ è in $O(2^n)$ e in $\Omega(\sqrt[b]{n})$.

  - $n^a = O(b^n)$ per ogni $a \geq 1$ ed ogni $b \geq 2$
  - $n^a = \Omega(\sqrt[b]{n})$ per ogni $a, b \geq 1$
  - Dunque, possiamo dire che **un polinomio è dominato da qualunque esponenziale** e che **un polinomio domina qualunque radice**

In termini più generali, notiamo come la nostra **scala di O grandi ed Omega** segua la **scala degli ordini di grandezza** delle successioni numeriche (per $n \to +\infty$):

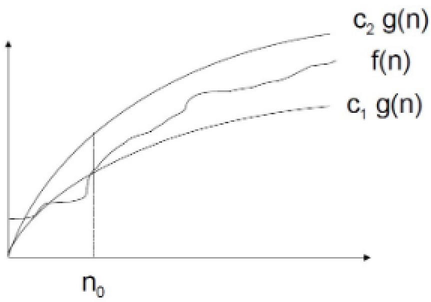
$$1 \prec log^a(n) \prec \sqrt[b]{n} \prec n^c \prec d^n \prec n! \prec n^n$$

2.1.4 Notazione Teta

Una volta definite le **notazioni O grande ed Omega**, possiamo dare una definizione di **notazione Teta**:

Definition 6. Teta

Date due funzioni $f(n), g(n) \geq 0$ si dice che ***f(n)* è in Θ*(g(n))*** se esistono tre costanti c_1, c_2 ed n_0 tali che $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ per ogni $n \geq n_0$



Dunque, se $f(n)$ è **sia** in $O(g(n)$ **sia** in $\Omega(g(n))$, allora è anche in $\Theta(g(n))$.

La **notazione Teta**, quindi, rappresenta il **limite stretto asintotico** della funzione: una volta superata una certa n, la funzione $f(n)$ **si comporta come** $g(n)$.

2.1.5 Calcolo delle Notazioni Asintotiche con i Limiti

In termini generali, possiamo formulare le seguenti regole basandoci sulla definizione di limite per $n \to +\infty$:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = k > 0 \text{ allora } f(n) = \Theta(g(n))$$

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty \text{ allora } f(n) = \Omega(g(n)) \text{ \underline{ma} } f(n) \neq \Theta(g(n))$$

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0 \text{ allora } f(n) = O(g(n)) \text{ \underline{ma} } f(n) \neq \Theta(g(n))$$

Se il limite del rapporto tra $f(n)$ e $g(n)$ **non esiste**, allora è necessario procedere diversamente.

2.2 Algebra della Notazione Asintotica

Oltre all'uso del calcolo con limiti, per semplificare il **calcolo del costo computazionale** tramite limite asintotico degli algoritmi possono essere utilizzate **tre regole algebriche**:

- **Regola delle costanti moltiplicative**
- **Regola della commutatività con somma**
- **Regola della commutatività con prodotto**

Le **dimostrazioni** di tali regole possono essere facilmente ricavate dalle definizioni stesse dei vari limiti asintotici, tuttavia verranno omesse poiché non utili ai fini dell'apprendimento.

Definition 7. Regola delle Costanti Moltiplicative

- Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è in $O(g(n))$ allora anche $k \cdot f(n)$ è in $O(g(n))$.
- Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è in $\Omega(g(n))$ allora anche $k \cdot f(n)$ è in $\Omega(g(n))$.
- Per ogni $k > 0$ e per ogni $f(n) \geq 0$, se $f(n)$ è in $\Theta(g(n))$ allora anche $k \cdot f(n)$ è in $\Theta(g(n))$.

In modo informale, quindi, possiamo affermare che **le costanti moltiplicative possono essere ignorate** durante il calcolo di un qualsiasi limite asintotico.

ATTENZIONE: è necessario sottolineare che è necessario che la costante moltiplicativa **non sia all'esponente** della funzione (es: nella funzione $f(n) = 2^{k*n}$ non possiamo ignorare la k)

Definition 8. Regola della Commutatività con Somma

Sia $f(n) = p(n) + q(n)$. Per ogni $p(n), q(n) > 0$ abbiamo che:

- Se $p(n)$ è in $O(g(n))$ e $q(n)$ è in $O(h(n)) \implies f(n)$ è in $O(g(n) + h(n)) = O(max(g(n), h(n)))$.
- Se $p(n)$ è in $\Omega(g(n))$ e $q(n)$ è in $\Omega(h(n)) \implies f(n)$ è in $\Omega(g(n) + h(n)) = \Omega(max(g(n), h(n)))$.
- Se $p(n)$ è in $\Theta(g(n))$ e $q(n)$ è in $\Theta(h(n)) \implies f(n)$ è in $\Theta(g(n) + h(n)) = \Theta(max(g(n), h(n)))$.

In modo informale, quindi, possiamo affermare che data una funzione $f(n) = p(n) + q(n)$, allora un qualsiasi limite asintotico di $f(n)$ è uguale al **massimo tra il limite asintotico di $p(n)$ e di $q(n)$**

Definition 9. Regola della Commutatività con Prodotto

Sia $f(n) = p(n) \cdot q(n)$. Per ogni $p(n), q(n) > 0$ abbiamo che:

- Se $p(n)$ è in $O(g(n))$ e $q(n)$ è in $O(h(n)) \implies f(n)$ è in $O(g(n) \cdot h(n))$.
- Se $p(n)$ è in $\Omega(g(n))$ e $q(n)$ è in $\Omega(h(n)) \implies f(n)$ è in $\Omega(g(n) \cdot h(n))$.
- Se $p(n)$ è in $\Theta(g(n))$ e $q(n)$ è in $\Theta(h(n)) \implies f(n)$ è in $\Theta(g(n) \cdot h(n))$.

In modo informale, quindi, possiamo affermare che data una funzione $f(n) = p(n) \cdot q(n)$, allora un qualsiasi limite asintotico di $f(n)$ è uguale al **prodotto tra il limite asintotico di $p(n)$ e di $q(n)$**

Esercizi svolti sull'algebra asintotica

1. Trovare il limite asintotico stretto di $f(n) = 3n^2 + 7$
$$f(n) = 3n^2 + 7 = \Theta(n^2) + \Theta(1) = \Theta(n^2)$$
2. Trovare il limite asintotico stretto di $f(n) = 3n2^n + 4n^4$
$$f(n) = 3n2^n + 4n^4 = \Theta(n2^n) + \Theta(n^4) = \Theta(n2^n)$$
3. Trovare il limite asintotico stretto di $f(n) = 2^{2n}$
$$f(n) = 2^{2n} = 2^n \cdot 2^n = \Theta(2^n) \cdot \Theta(2^n) = \Theta(2^{2n})$$
4. Trovare il limite asintotico stretto di $f(n) = log^n(n) + 8 \cdot 2^{n \cdot log(n)} + 3$
$$f(n) = log^n(n) + 8 \cdot 2^{n \cdot log(n)} + 3 = log^n(n) + 8 \cdot 2^{log(n^n)} + 3 =$$
$$= log^n(n) + 8 \cdot n^n + 3 = \Theta(log^n(n)) + \Theta(n^n) + \Theta(1) = \Theta(n^n)$$

5. Trovare il limite asintotico stretto di $f(n) = n^2 \cdot log(n)$

$$f(n) = n^2 \cdot log(n) = \Theta(n^2) \cdot \Theta(log(n)) = \Theta(n^2 \cdot log(n))$$

6. Trovare il limite asintotico stretto di $f(n) = 3n \cdot log(n) + 2n^2$

$$f(n) = 3n \cdot log(n) + 2n^2 = \Theta(n) \cdot \Theta(log(n)) + \Theta(n^2) = \Theta(n \cdot log(n)) + \Theta(n^2) = \Theta(n^2)$$

7. Trovare il limite asintotico stretto di $f(n) = 2^{\frac{log(n)}{2}} + 5n$

$$f(n) = 2^{\frac{log(n)}{2}} + 5n = (2^{log(n)})^{\frac{1}{2}} + 5n = \sqrt{n} + 5n = \Theta(\sqrt{n}) + \Theta(n) = \Theta(n)$$

8. Trovare il limite asintotico stretto di $f(n) = 4^{log(n)}$

$$f(n) = 4^{log(n)} = (2^2)^{log(n)} = (2^{log(n)})^2 = n^2 = \Theta(n^2)$$

9. Trovare il limite asintotico stretto di $f(n) = \sqrt{2}^{log(n)}$

$$f(n) = \sqrt{2}^{log(n)} = 2^{\frac{1}{2} \cdot log(n)} = \sqrt{2^{log(n)}} = \sqrt{n} = \Theta(\sqrt{n})$$

2.2.1 Sommatorie e Tecniche di dimostrazione

Di seguito vedremo alcune **tecniche di dimostrazione** che ci permettono di stabilire se una certa **funzione semplice o complessa** rientri all'interno di una determinata famiglia di funzioni O grandi, Omega o Teta.

- Dimostrare o confutare la seguente proposizione

$$f(n) = 4^n \text{ è in } O(2^n)$$

Tramite l'algebra asintotica, siamo già in grado di rispondere a tale proposizione:

$$4^n = 2^{2n} = 2^n \cdot 2^n = O(2^n) \cdot O(2^n) = O(2^{2n})$$

Dunque, la proposizione è **falsa**. Tuttavia, scegliamo di dimostrare la cosa in forma più rigorosa:

Dimostrazione per assurdo: supponiamo che $f(n) = O(2^n)$. Allora abbiamo che

$$\exists c, n_0 \mid f(n) \leq c \cdot 2^n, \ \ \forall n \geq n_0$$

$$4^n \leq c \cdot 2^n$$

$$2^n \cdot 2^n \leq c \cdot 2^n$$

$$2^n \leq c$$

Falso una volta superato un certo valore n_0

- Dimostrare la seguente proposizione

$$f(n) = (n + 10)^3 \text{ è in } \Theta(n^3)$$

Per dimostrare che sia $\Theta(n^3)$, dimostriamo che sia in $O(n^3)$ e in $\Omega(n^3)$:

- Ponendo $n_0 = 10$ abbiamo

$$(n + 10)^3 \leq (n + n)^3 = (2n)^3 = 8n^3 = O(n^3)$$

- Ponendo $n_0 = 0$ abbiamo

$$(n + 10)^3 \leq (n + 0)^3 = n^3 = \Omega(n^3)$$

Poiché $f(n)$ è **sia** in $O(n^3)$, **sia** in $\Omega(n^3)$, allora è **anche** in $\Theta(n^3)$

- Dimostrare la seguente proposizione

$$S_n = \sum_{k=1}^{n} k \text{ è in } \Theta(n^2)$$

Come nell'esempio precedente, per dimostrare che sia $\Theta(n^2)$, dimostriamo che sia in $O(n^2)$ e in $\Omega(n^2)$:

- Per dimostrare che S_n è in $O(n^2)$, è necessario fare un **"salto logico"**. Partiamo riscrivendo la sommatoria in forma estesa

$$S_n = 1 + 2 + 3 + ... + (n - 1) + n$$

Notiamo come **ogni singolo termine della sommatoria sia** $\leq n$. Dunque, possiamo scrivere la seguente disequazione:

$$1 + 2 + 3 + ... + (n - 1) + n \leq n + n + n + ... + n + n$$

Nella parte destra della disequazione, dunque, abbiamo una **somma di** n **volte** n, riscrivibile come $n \cdot n$

$$S_n \leq n \cdot n \Longrightarrow S_n \leq n^2$$

A questo punto, ci basta notare che n^2 **è in** $O(n^2)$ e quindi che, poiché $S_n \leq n^2$, di **conseguenza** anche S_n **è in** $O(n^2)$.

– Dimostriamo ora che $f(n) = \Omega(n^2)$ in modo analogo a quello precedente. Riscriviamo nuovamente la sommatoria in forma estesa

$$S_n = 1 + 2 + 3 + ... + (n-2) + (n-1) + n$$

Questa volta, notiamo che essa può essere **divisa a metà**, ottenendo **due categorie**:

$$\underbrace{1+2+3+4+5+...}_{\text{Numeri } \leq \frac{n}{2}} \quad \Big| \quad \underbrace{...+(n-2)+(n+1)+n}_{\text{Numeri } \geq \frac{n}{2}}$$

A questo punto, è necessario effettuare un **ulteriore "salto logico"**: sappiamo che **tutti i numeri minori di $\frac{n}{2}$ sono anche maggiori di 0**, mentre **quelli maggiori di $\frac{n}{2}$ sono ovviamente maggiori di $\frac{n}{2}$**.

Dunque, possiamo scrivere la seguente disequazione:

$$1+2+3+...+(n-2)+(n-1)+n \geq \underbrace{0+0+0+...}_{\frac{n}{2} \text{ volte}} + \underbrace{...+\frac{n}{2}+\frac{n}{2}+\frac{n}{2}}_{\frac{n}{2} \text{ volte}}$$

$$S_n \geq \frac{n}{2} \cdot \frac{n}{2} \Longrightarrow S_n \geq \frac{1}{2}n^2$$

A questo punto, ci basta notare che $\frac{1}{2}n^2$ **è in** $\Omega(n^2)$ e quindi che, poiché $S_n \geq \frac{1}{2}n^2$, di **conseguenza** anche S_n **è in** $\Omega(n^2)$.

– Poiché S_n è **sia** in $O(n^2)$, **sia** in $\Omega(n^2)$, allora è **anche** in $\Theta(n^2)$

- Vediamo ora un ulteriore modo per poter dimostrare tale proposizione:

$$S_n = \sum_{k=1}^{n} k \text{ è in } \Theta(n^2)$$

– Anche in questa dimostrazione, riscriviamo nuovamente la sommatoria in forma estesa, ma anche in **forma invertita**:

$$S_n = 1 + 2 + 3 + ... + (n-2) + (n-1) + n$$
$$S_n = n + (n-1) + (n-2) + ... + 3 + 2 + 1$$

– Sommando S_n con **se stessa**, otteniamo il seguente risultato:

S_n	1	2	3	......	$n-2$	$n-1$	n
S_n	n	$n-1$	$n-2$	......	3	2	1
$2S_2$	$n+1$	$n+1$	$n+1$	......	$n+1$	$n+1$	$n+1$

Dunque, $2S_n = (n+1) + (n+1) + ... + (n+1) + (n+1) = n(n+1)$

- A questo punto, ci basta sfruttare alcune **proprietà algebriche**

$$2S_n = n(n+1)$$
$$S_n = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = \Theta(n^2)$$

- Dimostrare la seguente proposizione

$$S_n = \sum_{k=0}^{n} 2^k \text{ è in } \Theta(2^n)$$

  - Riscriviamo la somma in forma estesa per poi **moltiplicarla per 2**

$$S_n = 1 + 2 + 2^2 + 2^3 + ... + 2^n$$
$$2 \cdot S_n = 2 \cdot (1 + 2 + 2^2 + 2^3 + ... + 2^{n-1} + 2^n)$$
$$2S_n = 2 + 2^2 + 2^3 + 2^4 + ... + 2^n + 2^{n+1}$$

  - Gli unici termini **non condivisi** tra S_n e $2S_n$ sono 1 e 2^{n+1}

$$2S_n = 2 + 2^2 + 2^3 + 2^4 + ... + 2^n + 2^{n+1}$$

$$S_n = 1 + 2 + 2^2 + 2^3 + ... + 2^n$$

  - Dunque otteniamo che

$$S_n = 2S_n - S_n = 2^{n+1} - 1$$

  - A questo punto calcoliamo il **limite asintotico** del risultato

$$S_n = 2^{n+1} - 1 = 2^n \cdot 2 - 1 = \Theta(2^n) + \Theta(-1) = \Theta(2^n)$$

- Dimostrare la seguente proposizione

$$\sum_{k=1}^{n} k \cdot 2^k \text{ è in } \Theta(n \cdot 2^n)$$

  - Riscriviamo in forma estesa e moltiplichiamo per 2

$$S_n = 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + ... + (n-1) \cdot 2^{n-1} + n \cdot 2^n$$
$$2S_n = 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + ... + (n-1) \cdot 2^n + n \cdot 2^{n+1}$$

– Effettuiamo qualche passaggio algebrico

$$S_n = 2S_n - S_n = -2^1 - 2^2 - 2^3 - 2^4 - ... - 2^{n-1} + n \cdot 2^{n+1}$$

$$-S_n = -(-2^1 - 2^2 - 2^3 - 2^4 - ... - 2^n + n \cdot 2^{n+1})$$
$$-S_n = 2^1 + 2^2 + 2^3 + 2^4 + ... + 2^n - n \cdot 2^{n+1}$$

– A questo punto, è necessario ricordarsi che nella dimostrazione precedente abbiamo ottenuto che

$$\sum_{k=0}^{n} 2^k = 1 + 2^1 + 2^2 + 2^3 + 2^4 + ... + 2^n = 2^{n+1} - 1$$

dunque, possiamo riscrivere $-S_n$ come

$$-S_n = 2^1+2^2+2^3+2^4+...+2^n-n\cdot 2^{n+1} = \left(\sum_{k=0}^{n} 2^k\right) - 1 - n\cdot 2^{n+1} = 2^{n+1} - 2 - n\cdot 2^{n+1}$$

per poi calcolare S_n

$$-(-S_n) = -(2^{n+1} - 2 - n \cdot 2^{n+1}) = -2^{n+1} + 2 + n \cdot 2^{n+1}$$

– Infine, calcoliamo il limite asintotico del risultato

$$S_n = -2^{n+1} + 2 + n \cdot 2^{n+1} = \Theta(2^n) + \Theta(2) + \Theta(n \cdot 2^n) = \Theta(n \cdot 2^n)$$

- Dimostrare la seguente proposizione

$$\sum_{k=1}^{n} log(k) \text{ è in } \Theta(n \cdot log(n)$$

– Riscriviamo la sommatoria in forma estesa in modo da applicare le proprietà dei logaritmi

$$S_n = log(1) + log(2) + log(3) + ... + log(n) =$$
$$= log(1 \cdot 2 \cdot 3 \cdot ... \cdot n) = log(n!)$$

– Dunque, ignorando la costante c, verifichiamo l'ipotesi

$$log(n!) \leq n \cdot log(n)$$

$$log(n!) \leq log(n^n)$$

$$n! \leq n^n$$

– Estendendo il fattoriale, possiamo mettere in evidenza due categorie di numeri.

$$\underbrace{1 \cdot 2 \cdot 3 \cdot ...}_{\text{Numeri} \leq \frac{n}{2}} \cdot \underbrace{... \cdot (n-2) \cdot (n-1) \cdot n}_{\text{Numeri} \geq \frac{n}{2}} \leq n^n$$

Quindi, sappiamo che **tutti i numeri minori di $\frac{n}{2}$ sono anche maggior di 1**, mentre tutti i numeri maggiori di $\frac{n}{2}$ sono maggiori di $\frac{n}{2}$.

Dunque possiamo scrivere la seguente disequazione

$$\underbrace{1 \cdot 1 \cdot 1 \cdot ...}_{\frac{n}{2} \text{ volte}} \cdot \underbrace{... \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2}}_{\frac{n}{2} \text{ volte}} \leq 1 \cdot 2 \cdot 3 \cdot ... \cdot (n-2) \cdot (n-1) \cdot n \leq n^n$$

$$1^{\frac{n}{2}} \cdot \left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

A questo punto, ri-applichiamo nuovamente il logaritmo ad ogni componente della disequazione

$$log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \leq log(n!) \leq log(n^n)$$

$$\frac{n}{2} log\left(\frac{n}{2}\right) \leq log(n!) \leq n \cdot log(n)$$

$$\frac{n}{2}(log(n) - log(n)) \leq log(n!) \leq n \cdot log(n)$$

$$\Theta(n \cdot log(n) \leq log(n!)) \leq \Theta(n \cdot log(n))$$

– Poiché S_n si trova **tra due** funzioni in $\Theta(n \cdot log(n))$, ne segue che **anche esso sia in** $\Theta(n \cdot log(n))$

Dunque, abbiamo visto come in molti casi **non sia sufficiente conoscere solo le regole dell'algebra asintotica**, soprattutto nel caso delle **sommatorie**.

Esercizi svolti

- Dimostrare che $f(n) = 4^n$ è in $O(2^{n \cdot log(n)})$

  Dimostrazione per assurdo: supponiamo che $f(n) \neq O(2^{n \cdot log(n)})$. Per definizione, ne segue che $\exists c, n_0 \mid f(n) \geq c \cdot 2^{n \cdot log(n)}, \ \ \forall n \geq n_0$

$$4^n \geq c \cdot 2^{n \cdot log(n)}$$

$$4^n \geq c \cdot n^n$$

$$log(2^{2n}) \geq log(c \cdot n^n)$$

$$2 \geq \frac{log(c)}{n} + log(n)$$

  Falso una volta superato un certo valore di n_0. Dunque, ne segue che $f(n)$ **è in** $O(2^{n \cdot log(n)})$.

- Dimostrare che $f(n) = (n-50)^2$ è in $\Theta(n^2)$
  - $f(n)$ è in $O(n^2)$ poiché ponendo $n_0 = -50$ si ha

$$(n-50)^2 \leq (n+n)^2 = O(n^2)$$

  - $f(n)$ è in $\Omega(n^2)$ poiché ponendo $n_0 = 0$ si ha

$$(n-50)^2 \leq (n+0)^2 = \Omega(n^2)$$

  - Dunque, ne segue che $f(n) = \Theta(n^2)$
- Dimostrare che per $c \in \mathbb{N}$ vale

$$\sum_{k=0}^{n} k^c \text{ è in } O(n^{c+1})$$

  - Basta riscrivere la sommatoria ed evidenziare che tutti i numeri sono $\leq n^c$

$$S_n = 1^c + 2^c + 3^c + ... + n^c \leq n^c + n^c + n^c + ... + n^c = n \cdot n^c$$

$$n \cdot n^c = n^{c+1} = O(n \cdot n^c)$$

$$S_n = O(n^{c+1}) \text{ poiché } n^{c+1} = O(n^{c+1}) \text{ e } S_n \leq n^{c+1}$$

- Dimostrare che per $c \neq 1$ vale

$$\sum_{k=0}^{n} c^k \text{ è in } O(n \cdot c^n)$$

  - Basta riscrivere la sommatoria ed evidenziare che tutti i numeri sono $\leq c^n$

$$S_n = c^1 + c^2 + c^3 + ... + c^n \leq c^n + c^n + c^n + ... + c^n = n \cdot c^n$$

$$n \cdot c^n = O(n \cdot c^n)$$

$$S_n = O(n \cdot c^n) \text{ poiché } \leq n^{c+1} = O(n \cdot c^n) \text{ e } S_n \leq n \cdot c^n$$

Capitolo 3

Costo Computazionale

3.1 Valutazione del costo computazionale

Fino ad ora, abbiamo parlato di costo computazionale di funzioni ipotetiche. In questo capitolo vedremo come **calcolare effettivamente il costo computazionale di un algoritmo**, adottando il criterio della misura del costo uniforme.

Il costo computazionale, inteso come funzione che rappresenta il tempo di esecuzione di un algoritmo, sia una **funzione monotona non decrescente** in base alla dimensione dell'input. (poiché ovviamente non è possibile che aumentando l'input diminuisca il tempo di esecuzione)

Dunque, poiché il tempo di esecuzione è strettamente dipendente dalla **quantità di dati in input**, è prima necessario trovare all'interno del codice il **parametro** corrispondente ad esso:

- In un **algoritmo di ordinamento** esso sarà il numero di dati da ordinare;
- In un **algoritmo che lavora su una matrice** sarà il numero di righe e di colonne (quindi, 2 parametri);
- In un **algoritmo che opera su alberi** sarà il numero di nodi che compongono l'albero
- In altri casi, invece, l'individuazione del parametro è meno scontata.

La **notazione asintotica** è alla base del calcolo del costo computazionale degli algoritmi. Dunque, in base alla sua definizione stessa, tale costo computazionale potrà essere ritenuto valido solo **asintoticamente**, ossia considerando **input molto grandi**.

Difatti, esistono degli algoritmi che per dimensioni dell'input relativamente piccole hanno un comportamento diverso rispetto a quello per dimensioni grandi, perciò è opportuno considerare solo input grandi per calcolarne la vera efficienza.

Normalmente, per poterne calcolare il costo, un algoritmo viene prima scritto in **pseudo-codice**, in modo che sia chiaro, sintetico e non ambiguo. Per comodità, in questo corso useremo direttamente il **linguaggio Python** per scrivere gli algoritmi.

3.2 Costo delle istruzioni

Principalmente, siamo in grado di individuare **tre categorie di istruzioni**:

- **Istruzioni elementari**: tutte le istruzioni con un **tempo di esecuzione costante**, ossia che non dipendono dalla dimensione dell'input (es: operazioni aritmetiche, lettura e scrittura di una variabile, valutazione di una condizione logica, stampa a video, ...). Poiché il loro tempo di esecuzione è costante, esse hanno un **costo computazionale pari a** $\Theta(1)$

  Ad esempio, le seguenti istruzioni hanno tutte costo $\Theta(1)$:

`var = 10`	$\Theta(1)$
`var += 10 * 10`	$\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$
`print("Il valore di var è:", var)`	$\Theta(1)$

- **Blocchi if/else**: hanno un costo pari alla **somma** tra il **costo della verifica della condizione** (solitamente $\Theta(1)$ poiché all'interno vi è una istruzione semplice) e il **massimo** tra i **costi complessivi** del **blocco if** e del **blocco else**.

  Ad esempio, il costo del blocco if/else sottostante è $\Theta(1)$, poiché:

  - Il costo della **verifica della condizione** è $\Theta(1)$
  - Il costo del **blocco if** è $\Theta(1) + \Theta(1) = \Theta(1)$
  - Il costo del **blocco else** è $\Theta(1)$
  - Il costo finale dell'**intero blocco** è:

  $$\text{CostoVerifica} + \max(\text{CostoIf, CostoElse}) = \Theta(1) + max(\Theta(1), \Theta(1)) = \Theta(1)$$

  ```
  if (a > b):
      a += b
      print("Il valore di a+b è", a)
  else:
      print("Il valore di a è", a)
  ```

- **Blocchi iterativi**: hanno un costo pari alla **somma effettiva**, dunque **non asintotica**, dei **costi di ciascuna iterazione**, compreso il costo di **verifica della condizione**.

  Ne consegue, quindi, che se **tutte le iterazioni** hanno lo stesso costo, allora il costo del blocco iterativo è pari al **prodotto del costo di una singola iterazione per il numero di iterazioni**.

  Inoltre, è opportuno sottolineare che la condizione viene valutata **una volta in più rispetto al numero delle iterazioni**, poiché l'ultima valutazione, che darà esito negativo, è quella che terminerà l'iterazione.

  Ad esempio, il costo del seguente blocco for è $\Theta(n)$, poiché:

  - La **verifica della condizione** ha costo pari a $\Theta(1)$
  - **Ogni iterazione** ha un costo pari a $\Theta(1)$
  - Il **numero di iterazioni** dipende strettamente dalla dimensione dell'array in input
  - Il costo finale dell'**intero blocco** sarà quindi:

  $$\text{NumIterazioni} \cdot \text{CostoIterazione} + \text{UltimaIter} = n \cdot \Theta(1) + \Theta(1) = \Theta(n)$$

```
sum = 0
for i in range(len(A)):     #n iterazioni + Θ(1) dell'ultima verifica
    sum += A[i]             Θ(1)
print("La somma degli elementi dell'array è", sum)
```

Il **costo dell'algoritmo nel suo complesso**, quindi, è pari alla **somma dei costi delle istruzioni che lo compongono**.

Tuttavia, per via di ciò un dato algoritmo potrebbe avere costi diversi a seconda dell'input, poiché un input particolarmente **vantaggioso** per i blocchi if/else e iterativi darebbe vita ad un **caso migliore**, mentre uno particolarmente **svantaggioso** darebbe vita ad un **caso peggiore.**

Dunque, per avere un'idea del costo di un algoritmo, preferiamo conoscere quale sia il suo comportamento nel **caso peggiore**. Ciò ci permette, quindi, di scegliere l'uso di un algoritmo rispetto ad un altro in previsione di **grandi quantità di input sfavorevoli**.

Per mantenere un'idea ottimale di precisione, tuttavia, utilizziamo comunque la **notazione Teta**, e non quella O grande. Laddove questo non sia possibile, **approssimeremo** il costo dell'algoritmo per difetto, dunque notazione Omega, o per eccesso, dunque notazione O grande.

3.3 Esempi di valutazione di un algoritmo

Esempio 1 - Calcolo del massimo di un array

Vediamo un primo esempio molto semplice. Proviamo ad analizzare l'algoritmo per il calcolo del massimo in un vettore disordinato contenente n valori:

```
def Trova_Max(A):
    n = len(A)                 Θ(1)
    max = A[1]                 Θ(1)
    for i in range (1,n):      #n − 1 iterazioni + Θ(1)
        if A[i] > max:         Θ(1)
            max = A[i]         Θ(1)
    return max                 Θ(1)
```

Procediamo in modo verticale, "suddividendo" il programma in **tre blocchi**: uno precedente al ciclo for nel mezzo, uno successivo ed uno corrispondente col ciclo for stesso:

- Costo **blocco precedente al blocco for**:

$$B_{PF} = \Theta(1) + \Theta(1) = \Theta(1)$$

- Costo del **blocco for**:

$$B_F = (n-1) \cdot \Theta(1) + \Theta(1) = \Theta(n-1) + \Theta(1) = \Theta(n) + \Theta(1) = \Theta(n)$$

  Attenzione: ricordiamo $\Theta(n-1) = \Theta(n)$ poiché prendiamo il massimo tra i due costi

- Costo **blocco successivo al for**:

$$B_{SF} = \Theta(1)$$

  Una volta calcolato il costo di ognuno dei tre blocchi, possiamo **sommare asintoticamente** (e non somma effettiva) i loro "costi parziali". Il costo complessivo dell'algoritmo sarà quindi:

- Costo **complessivo algoritmo**:

$$T(n) = B_{PF} + B_F + B_{SF} = \Theta(1) + \Theta(n) + \Theta(1) = \Theta(n)$$

Esempio 2 - Somma dei primi n interi

Vediamo ora un esempio di possibile ottimizzazione di un algoritmo: dato un numero n in input, vogliamo ottenere la somma di tutti i numeri a partire da 0 fino ad n. Analizziamo quindi il seguente algoritmo:

```
def Calcola_Somma_1(n):
    somma = 0                   Θ(1)
    for i in range (1,n+1):     #n iterazioni + Θ(1)
        somma += i              Θ(1)
    return somma                Θ(1)
```

Essendo una situazione molto simile all'esempio precedente, siamo in grado di calcolare facilmente il costo di questo algoritmo:

$$T(n) = \Theta(1) + [n \cdot \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$$

Tuttavia, come visto nella sezione 2.2.1, sappiamo che esiste un **metodo matematico** per calcolare **direttamente** la somma dei primi n numeri:

$$\sum_{k=0}^{n} 1 = \frac{n(n+1)}{2}$$

Possiamo quindi scrivere una versione **nettamente ottimizzata** dell'algoritmo, ottenendo un costo pari a $\Theta(1)$, poiché vengono effettuate **solo istruzioni semplici** di costo costante:

```
def Calcola_Somma_2(n):
    somma = n*(n+1)/2
    return somma
```

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

Spesso, infatti, è possibile **ottimizzare** pezzi di algoritmo che si occupano di **calcolo matematico** con delle **formule dirette**, che permettono di ridurre notevolmente il costo dell'algoritmo.

Esempio 3 - Valutazione di un polinomio in un punto

Vediamo ora un esempio più complesso dei precedenti: vogliamo valutare un polinomio espresso nella seguente forma

$$\sum_{k=0}^{n} a_i x^i$$

dove ogni a_i corrisponde ad un elemento di un array dato in input assieme al valore assunto dalla variabile x.

Ad esempio, il seguente polinomio verrà espresso nell'array nella forma qui riportata

$$x^2 - 4x + 5 = [a_0, a_1, a_2] = [5, 4, 1]$$

```
def Calcola_Polinomio_1(A, x):
    somma=0                          Θ(1)
    for i in range(len(a)):          #n iterazioni + Θ(1)
        potenza = 1                  Θ(1)
        for j in range(i):           #i iterazioni + Θ(1)
            potenza = x*potenza      Θ(1)
        somma = somma+A[i]*potenza   Θ(1)
    return somma                     Θ(1)
```

Come possiamo notare, questa volta abbiamo una situazione contorta: vi sono **due cicli for annidati**, dove il **contatore del secondo** dipende dal **contatore del primo**.

Ciò significa che il **ciclo for interno** verrà eseguito prima 0 volte, poi 1, poi 2 e così via finché il contatore i del primo for non raggiungerà n. Poiché il costo di un ciclo for è costituito dalla **somma effettiva** (e non asintotica) dei costi delle sue iterazioni, questa casistica è perfettamente esprimibile tramite una **sommatoria**:

$$\sum_{i=0}^{n} \Theta(1)$$

Le **sommatorie**, nell'ambito della notazione asintotica, godono di una particolare proprietà: esse sono **intercambiabili con la notazione utilizzata**:

$$\sum_{i=0}^{n} \Theta(1) = \Theta\left(\sum_{i=0}^{n} 1\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Dunque, il costo finale dell'algoritmo sarà:

$$T(n) = \Theta(1) + \left(\sum_{i=0}^{n} (\Theta(1) + \Theta(1) + \Theta(1))\right) + \Theta(1) = \Theta(1) + \Theta(n^2) + \Theta(1) = \Theta(n^2)$$

Tuttavia, guardando meglio l'algoritmo notiamo al suo interno una **grande possibile ottimizzazione**: invece che ricalcolare la potenza corrispondente ad ogni termine del polinomio, possiamo **conservare** la potenza calcolata per il termine precedente, riducendo la necessità di dover utilizzare il secondo ciclo for:

```
def Calcola_Polinomio_2(A, x):
    somma=0                             Θ(1)
    potenza=1                           Θ(1)
    for i in range(len(a)):             #n iterazioni + Θ(1)
        potenza = x*potenza             Θ(1)
        somma = somma+A[i]*potenza      Θ(1)
    return somma                        Θ(1)
```

Il nuovo costo dell'algoritmo sarà quindi nettamente migliore della versione precedente:

$$T(n) = \Theta(1) + n \cdot \Theta(1) + \Theta(1) = \Theta(n)$$

Esempio 4 - Analisi del caso migliore e caso peggiore

Analizziamo il seguente algoritmo dove esistono un caso migliore ed un caso peggiore con un costo computazionale differente.

```
def es4(n):
    if n<0: n=-n
    while n:
        if n%2: return 1
        n-=2
    return 0
```

- Analisi del ciclo while:
  - Se n è **dispari**, allora la condizione $ifn\%2$ restituirà **True**, eseguendo l'istruzione **return** e terminando istantaneamente il ciclo.

    Dunque abbiamo un costo pari a $\Theta(1)$.

– Se n è **pari**, allora il comportamento del ciclo, sarà

n. Iterazione	1	2	3	4	...	k
Valore di n	$n-2$	$n-4$	$n-6$	$n-4$	...	$n-2k$

finché $n - 2k = 0$ (condizione necessaria a terminare il while).

Dunque, il costo sarà $\Theta(n)$

$$n - 2k = 0$$

$$n = 2k$$

$$k = \frac{n}{2}$$

$$\frac{1}{2} \cdot \Theta(n) = \Theta(n)$$

- Possiamo quindi dire che

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \text{ è dispari (caso migliore)} \\ \Theta(n) & \text{se } n \text{ è pari (caso peggiore)} \end{cases}$$

Esempio 5 - Iterazioni con radice

Vediamo ora un esempio in cui il numero di iterazioni del ciclo descritto corrisponde ad una radice:

```
def es5(n):
    n=abs(n)
    x=r=0
    while x*x<n:
        x+=1
        r*=3*x
    return r
```

Analisi del ciclo while:

- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Valore di x	1	2	3	4	...	k
Valore di $x \cdot x$	1^2	2^2	3^2	4^2	...	k^2

finché $x \cdot x = n$ (condizione necessaria a terminare il while).

Dunque, il numero di iterazioni sarà

$$x \cdot x = n$$
$$x^2 = n$$
$$x = \sqrt{n}$$

- Costo finale:

$$T(n) = \Theta(1) + \sqrt{n} \cdot \Theta(1) + \Theta(1) = \Theta(\sqrt{n})$$

Esempio 6 - Iterazioni logaritmiche

Dopo aver visto un esempio con iterazioni con radice, vediamo un caso in cui otteniamo delle iterazioni logaritmiche

```
def es6(n):
    n=abs(n)
    x=r=0
    while n>1:
        r+=2
        n=n//3
    return r
```

Analisi del ciclo while:

- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Valore di n	$\frac{n}{3}$	$\frac{n}{3^2}$	$\frac{n}{3^3}$	$\frac{n}{3^4}$	...	$\frac{n}{3^k}$

finché $\frac{n}{3^k} = 1$ (condizione necessaria a terminare il ciclo while).

Dunque, il numero di iterazioni sarà

$$\frac{n}{3^k} = 1$$
$$n = 3^k$$
$$k = log_3(n)$$

- Costo finale:

$$T(n) = \Theta(1) + log_3(n) \cdot \Theta(1) + \Theta(1) = \Theta(log(n))$$

Esempio 7 - Iterazioni esponenziali

Infine, per completezza vediamo un esempio con iterazioni esponenziali

```
def es7(n):
    n=abs(n)
    x=t=1
    for i in range(n):
        t=3*t
    while t>=x:
        x+=2
        t-=x
    return x
```

Analisi del ciclo for:

- Il ciclo viene eseguito n volte, dove ad ogni iterazione la variabile t viene moltiplicata per 3. Dunque, alla fine del ciclo avremo $t = 3^n$.

Analisi del ciclo while:

- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Valore di x	3	5	7	9	...	1+2k
Valore di t	$3^n - 3$	$3^n - 5$	$3^n - 7$	$3^n - 9$	...	$3^n - (1 + 2k)$

  finché $3^n - (1 + 2k) = 2k$ (condizione necessaria a terminare il while)

  Dunque, il numero di iterazioni sarà

$$3^n - 1 - 2k = 2k$$
$$3^n - 1 = 4k$$
$$k = \frac{3^n - 1}{4}$$

- Costo finale:

$$T(n) = \Theta(1) + n \cdot \Theta(1) + \frac{3^n - 1}{4} \cdot \Theta(1) + \Theta(1) = \Theta(n) + \Theta(3^n) = \Theta(3^n)$$

Esempio 8

```
def es8(n):
    n=abs(n)
    p=2
    while n>=p:
        p=p*p
    return p
```

Analisi del ciclo while:

- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Valore di p	2^2	2^4	2^6	2^8	...	2^{2^k}

  finché $2^{2^k} = n + 1$ (condizione necessaria a terminare il while)

  Dunque, il numero di iterazioni sarà

$$2^{2^k} = n + 1$$
$$2^k = log_2(n + 1)$$
$$k = log_2(log_2(n + 1))$$

- Costo finale:

$$T(n) = \Theta(1) + log_2(log_2(n + 1)) \cdot \Theta(1) + \Theta(1) = \Theta(log(log(n)))$$

Esempio 9

```
def es9(n):
    n=abs(n)
    i,j,t,s=1
    while i*i<=n:
        for j in range(t)
            s+=1
        i=i+1
```

```
        t+=1
    return s
```

Analisi del ciclo while:

- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Valore di t	2	3	4	5	...	k+1
Valore di i	2	3	4	5	...	k+1
Valore di i^2	2^2	3^2	4^2	5^2	...	$(k+1)^2$

  finché $(k+1)^2 = n+1$ (condizione necessaria a terminare il while)

  Dunque, il numero di iterazioni sarà

$$\begin{aligned}(k+1)^2 &= n+1\\ k+1 &= \sqrt{n+1}\\ k &= \sqrt{n+1}-1\end{aligned}$$

Analisi del ciclo for annidato:

- Ad ogni iterazione del ciclo while, il ciclo for viene eseguito t volte. Tuttavia, il valore di t aumenta di 1 ad ogni iterazione del while, dunque il numero di iterazioni sarà

$$\sum_{t=1}^{\sqrt{n+1}-1} t = \frac{(\sqrt{n+1}-1)\cdot\sqrt{n+1}}{2} = \frac{n+1-\sqrt{n+1}}{2}$$

- Costo finale:

$$T(n) = \Theta(1) + \frac{n+1-\sqrt{n+1}}{2}\cdot\Theta(1) + \Theta(1) = \Theta(n)$$

Esempio 10

```
def es10(n):
    n=abs(n)
    s=n
    p=2
    i,r=1
```

```
while s>=1:
    s=s//5
    p+=2
p=p*p
while i*i*i<n:
    for j in range(p):
        r+=1
    i+=1
return r
```

Analisi del primo ciclo while:

- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Valore di p	4	6	8	10	...	2+2k
Valore di s	$\frac{n}{5}$	$\frac{n}{5^2}$	$\frac{n}{5^3}$	$\frac{n}{5^4}$	...	$\frac{n}{5^k}$

  finché $\frac{n}{5^k} = 1$ (condizione necessaria a terminare il while)

  Dunque, il numero di iterazioni sarà

$$\frac{n}{5^k} = 1$$
$$n = 5^k$$
$$k = log_5(n)$$

- Comportamento di p:

  Poiché il primo ciclo while viene eseguito $log_5(n)$ volte, anche l'istruzione `p += 2` viene eseguita $log_5(n)$ volte.

  Dunque, il valore di p, una volta concluso il primo ciclo while, sarà $p = 2+2log_5(n)$. Inoltre, viene eseguita l'istruzione `p = p * p`, dunque $p = (2 + 2log_5(n))^2$.

Analisi del secondo ciclo while:

- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Valore di i	1	2	3	4	...	k
Valore di i^3	2^3	3^3	4^3	5^3	...	$(k+1)^3$

  finché $(k + 1)^3 = n$ (condizione necessaria a terminare il while)

Dunque, il numero di iterazioni sarà

$$(k+1)^3 = n$$
$$k+1 = \sqrt[3]{n}$$
$$k = \sqrt[3]{n}+1$$

- Comportamento del ciclo for innestato:

  Viene eseguito ad ogni iterazione del ciclo while (dunque $\sqrt[3]{n}+1$ volte. Al suo interno, inoltre, vengono eseguite p iterazioni, dove il valore di p è $(2+2log_5(n))^2$.

- Costo finale:

$$T(n) = \Theta(1) + log_5(n)\cdot\Theta(1) + (\sqrt[3]{n}+1)((2+2log_5(n))^2\cdot\Theta(1)+\Theta(1)) + \Theta(1) =$$

$$= \Theta(log(n)) + (\sqrt[3]{n}+1)(\Theta(log^2(n))+\Theta(1)) = \Theta(log(n)) + \Theta(\sqrt[3]{n}\cdot log^2(n)) + \Theta(\sqrt[3]{n}) =$$
$$= \Theta(\sqrt[3]{n}\cdot log^2(n))$$

3.4 Tempi di esecuzione

Una volta appreso come poter valutare il costo di un algoritmo, possiamo effettivamente capire quanto sono grandi i **tempi di esecuzione** di un algoritmo in base al suo **costo computazionale**.

Ipotizziamo di disporre di un calcolatore in grado di effettuare una **operazione elementare in un nanosecondo** (dunque 10^9 operazioni al secondo) e supponiamo che la dimensione dei dati in input sia $n = 10^6$.

- Tempi di un algoritmo con costo $O(n)$

$$T = \frac{10^6}{10^9 \text{ op/s}} = 10^{-3} = 1 \text{ millisecondo}$$

- Tempi di un algoritmo con costo $O(n\cdot log(n))$

$$T = \frac{10^6\cdot log(10^6)}{10^9 \text{ op/s}} = \frac{3\cdot log(10)}{500} \approx 20 \text{ millisecondi}$$

- Tempi di un algoritmo con costo $O(n^2)$

$$T = \frac{(10^6)^2}{10^9 \text{ op/s}} = 10^3 = 1000 \text{ secondi} \approx 17 \text{ minuti}$$

Notiamo quindi che la differenza tra $O(n)$ e $O(n^2)$ è abissale. Ma che succede se il costo computazionale cresce esponenzialmente, ad esempio quando è $O(2^n)$?

È facile immaginare che il **tempo di esecuzione** esploda fino a raggiungere cifre astronomiche. Infatti, già con un input di dimensioni misere come $n = 100$, il tempo di esecuzione raggiunge una quantità inimmaginabile:

- Tempi di un algoritmo con costo $O(2^n)$

$$T = \frac{2^{100}}{10^9 \text{ op/s}} = 10^3 = 1,26 \cdot 10^{21} \text{ secondi} \approx 3 \cdot 10^{13} \text{ anni}$$

Dunque, possiamo concludere che un algoritmo di costo esponenziale sia **inutilizzabile**. Infatti, nonostante l'avanzamento tecnologico possa raggiungere potenzialità formidabili, non è in grado di rendere abbordabile la risoluzione di un tale problema.

Capitolo 4

Il Problema della Ricerca

Nell'informatica , esistono alcune tipologie di problemi particolarmente ricorrenti. In particolare, uno di essi è la **ricerca di un elemento in un insieme di dati** (es: numeri, cognomi, ...).

Tali problemi consistono in:

- **Input**: un **array** A di n elementi ed un **valore** v da cercare al suo interno
- **Output**: l'**indice** corrispondente alla posizione dell'elemento v trovato all'interno dell'array, oppure $Null$ o -1 se l'elemento non viene trovato

4.1 Ricerca sequenziale

La prima tipologia di **algoritmo di ricerca** è composta da tre passaggi:

- Presi in input il valore v da cercare e l'array, quest'ultimo viene analizzato **elemento per elemento**
- Ogni elemento viene **confrontato** con v. Se l'elemento analizzato e v **coincidono**, allora viene restituito l'indice dell'elemento, altrimenti si procede con il **prossimo elemento**
- Se anche l'**ultimo elemento** dell'array non coincide con v, allora viene restituito -1, indicando che l'elemento non è presente nella lista

Per via del suo funzionamento, tale algoritmo viene chiamato **Ricerca Sequenziale**.

Anche senza dover analizzare il codice, possiamo renderci facilmente conto del fatto che il **caso migliore** di tale algoritmo corrisponda al caso in cui l'elemento da cercare sia in **prima posizione** (ossia all'indice 0), mentre il **caso peggiore** corrisponda al caso in cui l'elemento **non sia presente** all'interno della lista (poiché comunque dovrebbe venir analizzata l'intera lista).

Dunque, possiamo già concludere che:

- **Caso migliore**: Se $v = A[0]$, allora $\Theta(1)$
- **Caso peggiore**: Se $v \notin A$, allora $\Theta(n)$

Poiché non abbiamo trovato una **stima del costo** che sia valida per tutti i casi, diremo che il **costo computazionale** dell'algoritmo è $O(n)$, per evidenziare il fatto che ci sono input in cui questo valore viene **raggiunto**, ma ci sono anche input in cui il costo è **minore**.

Una versione molto semplificata di questo algoritmo può essere implementata dal seguente **codice**:

```
def Ricerca_Sequenziale(A, v):
    i = 1
    while (i < len(A)) and (A[i] ≠ v):     # eseguito massimo n volte
        i += 1
    if i < len(A):
        return i
    else:
        return -1
```

4.1.1 Stima del costo medio

Come abbiamo visto, non è possibile determinare il costo asintotico stretto di tale algoritmo poiché il caso peggiore e il caso migliore sono differenti. In questi casi, è necessario effettuare una stima del **costo medio** dell'algoritmo, ossia quello che si verifica con **più probabilità**.

Ipotizziamo di avere un array A di n elementi al cui interno ogni posizione ha la **stessa probabilità** di contenere il valore v da cercare (dunque non ci sono posizioni favorite).

A questo punto, possiamo dire che la **probabilità che v sia in k-esima posizione** è

$$P = \frac{1}{n}$$

Applicando tale probabilità al **numero totale di iterazioni**, otteniamo

$$P \cdot \sum_{k=0}^{n} k = \frac{1}{2} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Dunque, possiamo dire che **in media** il ciclo viene eseguito $\frac{n+1}{2}$ volte, corrispondenti quindi ad un $\Theta(n)$. In questo caso, quindi, il **caso medio**, si **avvicina** più **al caso peggiore** rispetto che al caso minore.

In alternativa, il **costo medio** può essere trovato utilizzando il calcolo delle **permutazioni**.

Ricordando che le permutazioni di una lista (o parola) di n elementi (o lettere) corrispondono a $n!$, possiamo dire che le **permutazioni totali di A** sono $p_{tot} = n!$.

All'interno di questo insieme di permutazioni, vi sono anche i seguenti sotto-insiemi:

- Permutazioni in cui v è in **prima posizione**
- Permutazioni in cui v è in **seconda posizione**
- Permutazioni in cui v è in **terza posizione**
- ...
- Permutazioni in cui v è in **ultima posizione**

Ognuno di tali sotto-insiemi, corrisponde ad una **permutazione di** $n-1$ **elementi**, ossia $p_k = (n-1)!$. Dunque, il numero medio di iterazioni del ciclo corrisponderà a

$$\sum_{k=0}^{n}\left(k \cdot \frac{p_k}{p_{tot}}\right) = \sum_{k=0}^{n}\left(k \cdot \frac{(n-1)!}{n!}\right) = \sum_{k=0}^{n}\left(k \cdot \frac{1}{n}\right) = \frac{1}{n} \cdot \sum_{k=0}^{n} k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Anche in questo caso, dunque, concludiamo che il **numero medio di iterazioni del ciclo** è $\frac{n+1}{2}$, corrispondente ad un $\Theta(n)$.

4.1.2 Operatore *in* del linguaggio Python

Prima di procedere, è necessario mettere alla luce il vero comportamento dell'operatore "*in*" del linguaggio Python, la cui sintassi per l'utilizzo ricordiamo essere

```
<valore> in <lista>
```

Di seguito un esempio di codice per ricordare meglio il funzionamento:

```
if v in A:
    print("Il valore v è dentro A")
else:
    print("Il valore v non è dentro A")
```

Sulla superficie, tale istruzione può sembrare un semplice **costo** pari a $\Theta(1)$, poiché si tratta di una **singola istruzione**. Tuttavia, in realtà tale operatore corrisponde ad una scrittura **estremamente abbreviata** fornita dal linguaggio Python corrispondente ad una **ricerca sequenzale**, il cui costo medio sappiamo essere $\Theta(n)$.

4.2 Ricerca binaria

Nella vita di tutti i giorni, tuttavia, noi esseri umani non utilizziamo **mai** una ricerca di tipo sequenziale: se volessimo cercare una parola all'interno di un dizionario per saperne il significato, non leggeremmo mai l'intero dizionario parola per parola.

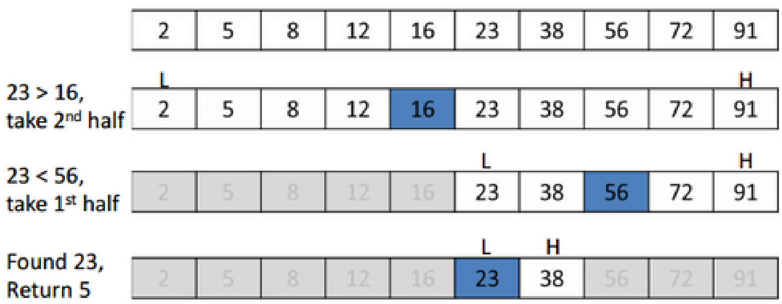
Ciò che il nostro cervello svolge (inconsciamente) è un'altra tipologia di algoritmo di ricerca, chiamata **ricerca binaria**: una volta aperto il dizionario ad una pagina casuale, abbiamo tre opzioni:

- La parola cercata si trova **nella pagina** che abbiamo aperto.
- La parola cercata si trova **prima della pagina** che abbiamo aperto. Dunque, il passo successivo sarà scegliere un'**altra pagina casuale all'interno solo delle pagine precedenti**.
- La parola cercata si trova **dopo la pagina** che abbiamo aperto. Dunque, il passo successivo sarà scegliere un'**altra pagina casuale all'interno solo delle pagine successive**.

Tale procedimento viene ripetuto fino a quando non troviamo la parola che stiamo cercando. Come sappiamo, tale ricerca ci risulta estremamente comoda e rapida, tant'è che il nostro cervello è in grado di sfruttarla senza fatica.

Possiamo quindi definire in modo più rigoroso l'algoritmo di **ricerca binaria**:

1. Viene ispezionato l'elemento centrale dell'array (che chiameremo m per comodità)
   - Se corrisponde al valore v che stiamo cercando (dunque $v = m$), viene restituito l'indice della posizione trovata
   - Se $v < m$, allora v si troverà nella **metà inferiore della lista**, dunque l'algoritmo verrà ripetuto solo su essa
   - Se $m < v$, allora v si troverà nella **metà superiore della lista**, dunque l'algoritmo verrà ripetuto solo su essa
2. **Ripetere** il passaggio finché la lista non si sarà ridotta ad **un solo elemento**
   - Se l'**unico elemento rimasto** dopo le riduzioni della lista effettuate corrisponde a v, allora verrà restituito l'indice trovato
   - Altrimenti, verrà restituito -1, indicando che l'elemento non è nella lista



*Un **esempio grafico** dell'algoritmo in cui viene ricercato il valore* 23

Tuttavia, è necessario sottolineare un **requisito necessario** per poter applicare l'algoritmo di ricerca binaria: al contrario della ricerca sequenziale, nella ricerca binaria **l'array deve essere <u>obbligatoriamente</u> ordinato in modo crescente**, altrimenti è impossibile utilizzare tale algoritmo.

Vediamo ora l'implementazione in codice di tale algoritmo:

```
def Ricerca_Binaria(A, v):
    a = 0     #primo indice di A
  b = len(A)-1     #ultimo indice di A
    m =(a+b)//2     #l'indice a metà di A
    while A[m] != v:
        if A[m] > v:
            b = m − 1     #prendo la metà inferiore
        else:
            a = m + 1     #prendo la metà superiore
        if a > b:     #si verifica solo se v non è in A
            return -1
        m=(a+b)//2     #calcolo di nuovo valore di m
    return m
```

Analisi del ciclo while:

- Ipotesi: supponiamo che l'elemento v venga trovato alla k-esima iterazione del ciclo
- Comportamento del ciclo:

n. Iterazione	1	2	3	4	...	k
Lunghezza di A	$\frac{n}{2}$	$\frac{n}{2^2}$	$\frac{n}{2^3}$	$\frac{n}{2^4}$	...	$\frac{n}{2^k}$

  finché $\frac{n}{2^k} = 1$ (condizione necessaria a terminare il while, poiché nel caso peggiore solo v rimane nell'array)

  Dunque, il numero di iterazioni sarà

$$\frac{n}{2^k} = 1$$
$$n = 2^k$$
$$k = log_2(n)$$

- Costo finale del caso peggiore:

$$T(n)_{peggiore} = \Theta(1) + log_2(n) \cdot \Theta(1) + \Theta(1) = \Theta(log(n))$$

Il costo computazionale del **caso peggiore** della **ricerca binaria**, quindi, è $\Theta(log(n))$, che è nettamente migliore rispetto al $\Theta(n)$ del caso peggiore della ricerca sequenziale.

Il **caso migliore**, invece, corrisponde ovviamente al caso in cui, appena avviata l'applicazione dell'algoritmo, si verifica che $v = A[m]$, dunque abbiamo un $\Theta(1)$

Come per la ricerca sequenziale, anche in questo caso il caso peggiore e il caso migliore discordano. Dunque, è necessario calcolare il **caso medio**.

Assunzioni:

- Il numero di elementi dell'array è una **potenza di 2** (per semplicità di calcolo)
- v **è presente nell'array** (dunque non si ricade nel caso peggiore)
- Tutte le posizioni dell'array hanno la **stessa probabilità** di contenere v

Analisi delle posizioni raggiungibili

1. Alla prima iterazione dell'algoritmo, le posizioni raggiungibili sono solo **una**, ossia quella centrale.
2. Alla seconda iterazione, le posizioni raggiungibili sono **due**:
   - Quella al centro della metà inferiore
   - Quella al centro della metà superiore
3. Alla terza iterazione, le posizioni raggiungibili sono **quattro**:
   - Quella al centro della metà inferiore della prima metà inferiore
   - Quella al centro della metà superiore della prima metà inferiore
   - Quella al centro della metà inferiore della prima metà superiore
   - Quella al centro della metà superiore della prima metà superiore
4. ...

Dunque, concludiamo che le **posizioni raggiungibili** da ogni **k-esima iterazione** sono $n(k) = 2^{k-1}$.

Poiché abbiamo assunto che ogni posizione è **equiprobabile**, dunque otteniamo che la probabilità di ogni posizione è

$$\frac{n(k)}{n} = \frac{2^{k-1}}{n}$$

Di conseguenza, il numero medio di iterazioni sarà

$$\sum_{k=0}^{log(n)} \left(k \cdot \frac{2^{k-1}}{n} \right) = \frac{1}{n} \cdot \sum_{k=0}^{log(n)} k \cdot 2^{k-1} = \frac{(log(n) - 1)2^{log(n)} + 1}{n} = log(n) - 1 + \frac{1}{n} = \Theta(log(n))$$

Capitolo 5

La ricorsione

Fino ad ora abbiamo trattato algoritmi in forma **iterativa**, ossia basati sull'uso di **cicli iterativi** (ciclo for, ciclo while, ...). In particolare, nel capitolo precedente, abbiamo visto una formulazione **iterativa** dell'algoritmo di **ricerca binaria**, dove veniva impiegato l'uso di un ciclo while. La ricerca binaria, tuttavia, è un esempio perfetto di caso in cui è possibile strutturare un algoritmo in modo **ricorsivo**.

Per capire cosa intendiamo, vediamo la seguente riformulazione ricorsiva dell'algoritmo:

- Ispeziona l'**elemento centrare** dell'array
- Se è **uguale** a v, restituisci il suo **indice**
- Se è **maggiore** di v, riduci l'array alla sua **metà inferiore** ed **ri-esegui la ricerca binaria** su di essa
- Se è minore di v, riduci l'array alla sua **metà superiore** e **ri-esegui la ricerca binaria** su di essa

L'aspetto cruciale di questa formulazione risiede nel fatto che l'**algoritmo risolve il problema "riapplicando" se stesso su un sotto-problema**, ossia una versione **"più semplice"** di esso (in questo caso su una delle due metà dell'array). Questa tecnica viene chiamata **ricorsione**.

Le **funzioni ricorsive** possono essere trovate anche nell'ambito matematico. Esempio tipico di ciò è il **fattoriale** di un numero:

$$n! = \begin{cases} n \cdot (n-1)! & \text{se } n > 0 \\ 1 & \text{se } n = 0 \end{cases}$$

$$n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1! \cdot 0! = n \cdot ... \cdot 1$$

Il fattoriale di n, dunque, è composto dal prodotto tra n stesso e il fattoriale di $(n-1)$, a sua volta composto dal prodotto tra $(n-1)$ e il fattoriale di $(n-2)$. Tale catena viene ripetuta fino a raggiungere il fattoriale di 0, che, per definizione, **equivale ad 1**.

Possiamo quindi considerare il fattoriale di 0 il **caso base** del problema, ossia il suo **sottoproblema minimo**, corrispondente alla versione più semplice possibile del problema stesso.

All'interno di un problema ricorsivo, dunque, la **catena di ricorsione viene ripetuta finché non viene raggiunto il caso base** del problema. Per questo motivo, è obbligatoria la presenza di **almeno un caso base** all'interno di un algoritmo ricorsivo, poiché in sua assenza la catena di ricorsione andrebbe avanti all'infinito.

Definition 10. Algoritmo ricorsivo

Un algoritmo è detto **ricorsivo** quando è espresso in termini di se stesso.

Un algoritmo ricorsivo ha sempre queste proprietà:

- La soluzione del problema complessivo è costruita risolvendo ricorsivamente uno o più **sottoproblemi di dimensione minore**, combinando le soluzioni ottenute
- la successione dei sottoproblemi, che sono sempre più piccoli, deve sempre convergere ad un sottoproblema che costituisca un **caso base**, il quale termina la ricorsione.

Esempi di codice ricorsivo

- Algoritmo ricorsivo per il calcolo del **fattoriale di** n

```
def fattoriale(n):     #n = intero non negativo
    if (n == 0) return 1
    return n * fattoriale(n - 1)
```

- Algoritmo ricorsivo della **ricerca sequenziale**

```
def Ricerca_seq_ric(A, v, n=len(A)-1):
    if (A[n]==v)
        return n
    if (n==0):
        return -1
    else:
        return Ricerca_seq_ric(A, v, n - 1)
```

- Algoritmo ricorsivo della **ricerca binaria**

```
def Ricerca_bin_ric (A, v, i_min=0, i_max=len(A)):
    if (i_min > i_max):
        return -1
    m =(i_min + i_max)//2
    if (A[m] = v):
        return m
    elif (A[m] > v):
        return Ricerca_bin_ric (A, v, i_min, m - 1)
    else:
        return Ricerca_bin_ric (A, v, m + 1, i_max)
```

5.1 Iterazione *vs* Ricorsione

Nella sezione precedente, abbiamo visto come sia possibile realizzare algoritmi iterativi anche in forma di algoritmi ricorsivi. Difatti, tale regola vale per ogni algoritmo: **qualsiasi problema risolvibile con un algoritmo ricorsivo può essere risolto anche con un algoritmo iterativo**.

Ma allora perché preferire la ricorsione all'iterazione e viceversa? Non potremmo semplicemente svolgere tutto in maniera iterativa? A tali domande è possibile rispondere con i tre seguenti punti:

- Utilizziamo un algoritmo ricorsivo quando si può formulare la soluzione del problema in un modo **aderente alla natura del problema stesso** (es: il fattoriale di un numero), mentre la soluzione iterativa è molto più complicata o addirittura non evidente
- Utilizziamo un algoritmo iterativo se esiste una soluzione iterativa altrettanto semplice e chiara quando paragonata alla sua versione ricorsiva
- Utilizziamo un algoritmo iterativo quando l'efficienza è un requisito primario

In particolare, bisogna porre attenzione sull'ultimo punto, poiché **ogni funzione**, sia essa ricorsiva o no, richiede per la sua esecuzione una certa **quantità di memoria**, per:

- Caricare in memoria il suo codice
- Passare i parametri e ritornare i valori calcolati
- Memorizzare i valori delle sue variabili locali

Dunque, poiché le **funzioni ricorsive** per loro natura stessa richiamano se stesse un elevato numero di volte, ne segue direttamente che esse abbiano un **consumo elevato in termini di memoria**.

Perciò, generalmente preferiamo l'iterazione alla ricorsione, a meno che il problema proposto non sia intuitivamente risolvibile come algoritmo ricorsivo, come nel seguente esempio:

- Progettare un algoritmo in grado di calcolare l'n-esimo numero di Fibonacci, definito come
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$ se $n > 1$

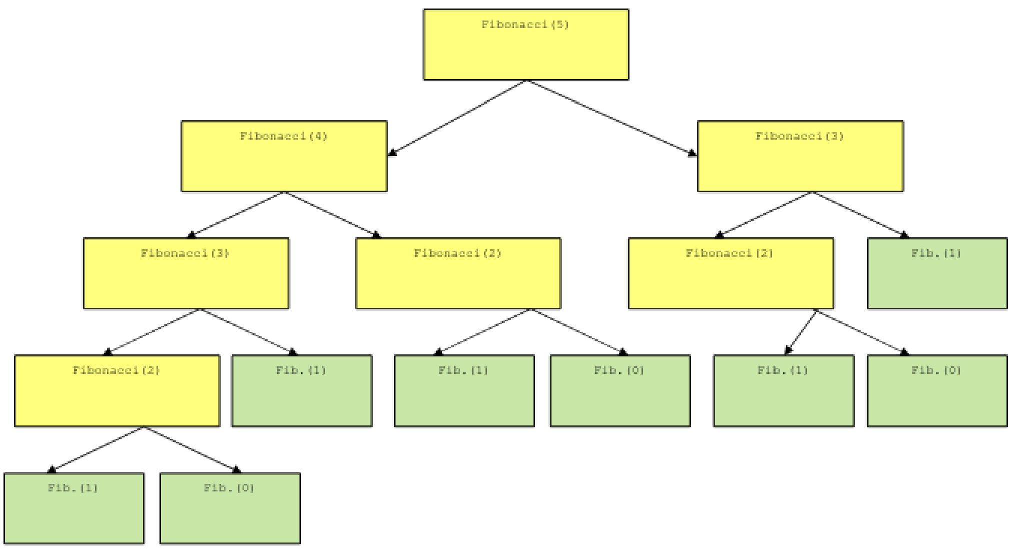
  In questo caso, viene naturale progettare l'algoritmo in termini di **ricorsione**, poiché l'n-esimo numero di Fibonacci viene calcolato tramite altri due numeri di Fibonacci.

  Il codice da implementare, dunque, risulta estremamente semplice nella sua **versione ricorsiva**:

```
def Fibonacci(n):
    if (n <= 1): return n
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

  Una **versione iterativa** del problema, invece, risulta particolarmente complessa da interpretare e sviluppare (si consiglia al lettore di effettuare un tentativo nella progettazione dell'algoritmo iterativo).

Analizziamo ora lo sviluppo della **catena di ricorsione** nel caso in cui volessimo calcolare il **quinto numero di Fibonacci**:



Notiamo velocemente alcune **osservazioni**:

- Nonostante l'input di piccole dimensioni, il **numero di chiamate ricorsive** effettuate è **enorme**, occupando un'elevata quantità di risorse
- Molti **calcoli** vengono **ripetuti** numerose volte ($Fib(1)$ viene eseguito 5 volte, $Fib(2)$ 3 volte, ...)

Di seguito viene proposta una **versione iterativa** dell'algoritmo:

```
def Fibonacci_iter(n):
    if (n <= 1): return n
    fib_prec_prec = 0
    fib_prec = 1
    for i in range(2,n+1):
        fib_prec_prec, fib_prec = fib_prec, fib_prec_prec + fib_prec
    return fib_prec
```

Siamo facilmente in grado di calcolare il **costo computazionale** di tale algoritmo, corrispondente a $\Theta(n)$. Ma qual è invece il costo della versione ricorsiva dell'algoritmo?

Considerando $T(n)$ come il costo computazionale della funzione, possiamo facilmente intuire che il **costo del primo sotto-problema** sarà $T(n-1)$, mentre quello del **secondo sotto-problema** sarà $T(n-2)$

```
def Fibonacci(n):
    if (n <= 1): return n                          Θ(1)
    return Fibonacci(n - 1) + Fibonacci(n - 2)     T(n-1) + T(n-2)
```

Il costo computazionale sarà quindi equivalente a

$$T = \begin{cases} T(n) = \Theta(1) + T(n-1) + T(n-2) & \text{se } n > 1 \\ T(1) = \Theta(1) & \text{se } n \leq 1 \end{cases}$$

Con gli strumenti attuali, non siamo ancora in grado di calcolare il costo effettivo di tale algoritmo, poiché necessario introdurre il concetto di **equazione di ricorrenza**.

Capitolo 6

Equazioni di ricorrenza

Come abbiamo visto, nell'ambito del calcolo del costo computazionale gli **algoritmi ricorsivi**, per loro natura stessa, danno vita ad una f**unzione matematica anch'essa ricorsiva**.

La funzione matematica ricorsiva che esprime il costo è anche detta **equazione di ricorrenza**.

Riprendiamo l'esempio del calcolo del fattoriale di un numero.

```
def fattoriale(n):
    if (n == 0) return 1              Θ(1)
    return n * fattoriale(n - 1)      T(n − 1)
```

Il costo computazionale di tale algoritmo risulta essere

$$T = \begin{cases} T(n) = \Theta(1) + T(n-1) & \text{se } n > 0 \\ T(0) = \Theta(1) & \text{se } n = 0 \end{cases}$$

La parte generale dell'equazione di ricorrenza che definisce $T(n)$ deve essere sempre costituita dalla **somma di almeno due addendi**, di cui **almeno uno contiene la parte ricorsiva** (nell'esempio $T(n-1)$, mentre uno rappresenta il costo computazionale di tutto ciò che viene eseguito al di fuori della chiamata ricorsiva (in questo caso il $\Theta(1)$). Inoltre, così come per l'algoritmo ricorsivo, anche nell'equazione di ricorrenza **deve sempre essere presente un caso base** (in questo caso $T(0)$).

Di seguito vedremo i **quattro metodi** utilizzabili per calcolare il **costo** di un'equazione di ricorrenza:

- **Metodo iterativo**
- **Metodo di sostituzione**
- **Metodo dell'albero**
- **Metodo principale**

6.1 Metodo iterativo

L'idea alla base del **metodo iterativo** è molto semplice: l'equazione di ricorrenza viene sviluppata in modo da essere espressa come **somma di termini** dipendenti dal **caso generico** e dal **caso base**. Per la sua natura stessa, tuttavia, tale metodo spesso risulta inefficiente per l'elevata quantità di calcoli da effettuare.

Consideriamo la seguente equazione di ricorrenza

$$T = \begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

Proviamo a **sviluppare** il termine $T(n)$ come **somma dei suoi sotto-termini**: se $T(n)$ è definito come $T(n-1) + \Theta(1)$, allora $T(n-1)$ sarà definito come $T(n-2) + \Theta(1)$ e così via

1. Per definizione abbiamo

$$T(n) = T(n-1) + \underbrace{\Theta(1)}_{\text{Una volta}} = T(n-2) + 1 \cdot \Theta(1)$$

2. Sviluppando $T(n-1)$, otteniamo che

$$T(n) = T(n-2) + \underbrace{\Theta(1) + \Theta(1)}_{\text{Due volte}} = T(n-2) + 2 \cdot \Theta(1)$$

3. Sviluppando $T(n-2)$, otteniamo che

$$T(n) = T(n-3) + \underbrace{\Theta(1) + \Theta(1) + \Theta(1)}_{\text{Tre volte}} = T(n-3) + 3 \cdot \Theta(1)$$

4. Sviluppando $T(n-(k-1))$, otteniamo che

$$T(n) = T(n-k) + \underbrace{\Theta(1) + \Theta(1) + ... + \Theta(1) + \Theta(1)}_{k \text{ volte}} = T(n-k) + k \cdot \Theta(1)$$

Abbiamo quindi ottenuto una **forma generalizzata** del caso generico $T(n)$ sviluppando i suoi sotto-termini k volte. Come sappiamo, però, dopo un **determinato numero di ricorsioni**, l'equazione di ricorrenza raggiungerà il suo **caso base** (nell'esempio $T(1)$), di cui sappiamo per certo il costo computazionale.

La catena di ricorsione, quindi, procederà finché $n - k = 1$, in modo che $T(n-k)$ corrisponda a $T(1)$. Dunque, da ciò ne segue che

$$n - k = 1 \Longrightarrow k = n - 1$$

Una volta trovato il valore di k, ci basterà sostituirlo all'interno dell'equazione per trovare il **costo computazionale generico**

$$\begin{gathered} T(n) = T(n-k) + k \cdot \Theta(1) = T(n-(n-1)) + (n-1) \cdot \Theta(1) = \\ = T(1) + \Theta(n) = \Theta(1) + \Theta(n) = \Theta(n) \end{gathered}$$

Ulteriori esempi

- Consideriamo la seguente equazione di ricorrenza

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

Sviluppando $T(n)$ otteniamo che

$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) = T\left(\frac{n}{2^2}\right) + \Theta(1) + \Theta(1) = T\left(\frac{n}{2^3}\right) + \Theta(1) + \Theta(1) + \Theta(1) = ...$

Generalizzando l'equazione, otteniamo

$$T(n) = T\left(\frac{n}{2^k}\right) + k \cdot \Theta(1)$$

Sappiamo che il caso base è $T(1)$ e che viene raggiunto quando

$$\frac{n}{2^k} = 1 \Longrightarrow k = log_2(n)$$

dunque ne segue che

$T(n) = T\left(\frac{n}{2^k}\right) + k \cdot \Theta(1) = T(1) + log_2(n) \cdot \Theta(1) = \Theta(1) + \Theta(log(n)) = \Theta(log(n))$

- Consideriamo la seguente equazione di ricorrenza

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

Sviluppando $T(n)$ otteniamo che

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1) = 2\left(2T\left(\frac{n}{2^2}\right) + \Theta(1)\right) + \Theta(1) =$$
$$= 2\left(2\left(2T\left(\frac{n}{2^2}\right) + \Theta(1)\right) + \Theta(1)\right) + \Theta(1) = ...$$

Generalizzando l'equazione, otteniamo

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \cdot \Theta(1)$$

Sappiamo che il caso base è $T(1)$ e che viene raggiunto quando

$$\frac{n}{2^k} = 1 \Longrightarrow k = log_2(n)$$

Dunque ne segue che

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \cdot \Theta(1) = 2^{log_2(n)} \cdot T(1) + \sum_{i=0}^{log_2(n)-1} 2^i \cdot \Theta(1) = \Theta(n) + \Theta\left(\frac{2^{log_2(n)} - 1}{2 - 1}\right) = \Theta(n)$$