SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA
FACOLTÀ DI INFORMATICA

Introduzione agli Algoritmi

Author
Simone Bianco

6 giugno 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 $n = 10^6$ 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**:

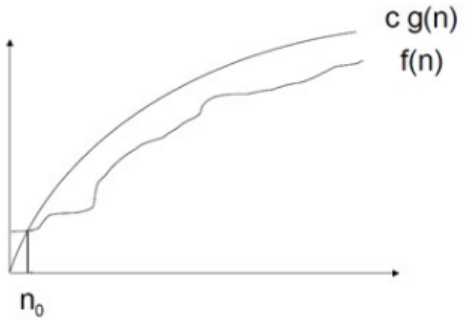
- **Notazione asintotica O grande**: definisce il limite superiore asintotico
- **Notazione asintotica** Ω: definisce il limite inferiore asintotico
- **Notazione asintotica** Θ: definisce il limite asintotico stretto

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).

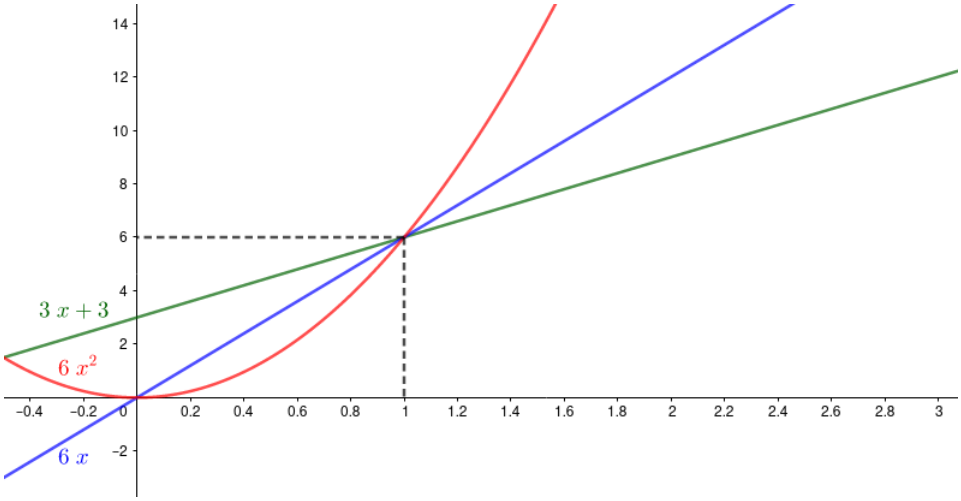
Esempi sull'O grande

- Sia $f(n) = 3n+3$. Possiamo dire che ***f(n)* è in *O(n^2)***, in quanto esiste una almeno una c (ossia $c = 6$) per cui :

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

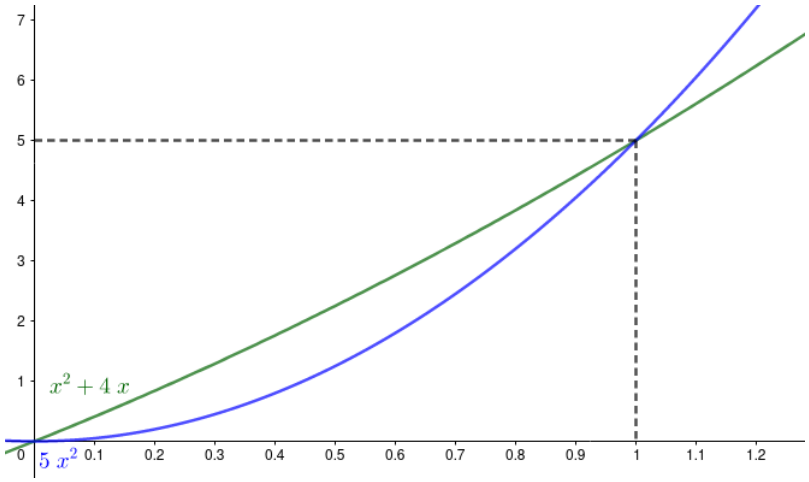
Tuttavia, possiamo anche dire che **f(n) è in O(n)**, in quanto:

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



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

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

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

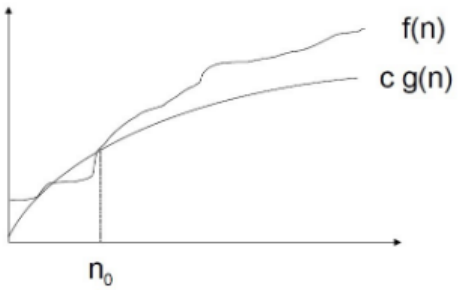
che per ipotesi sappiamo essere vera, dunque concludiamo che $f(n)$ è in $O(n^m)$

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 $\Omega(g(n))$** 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).

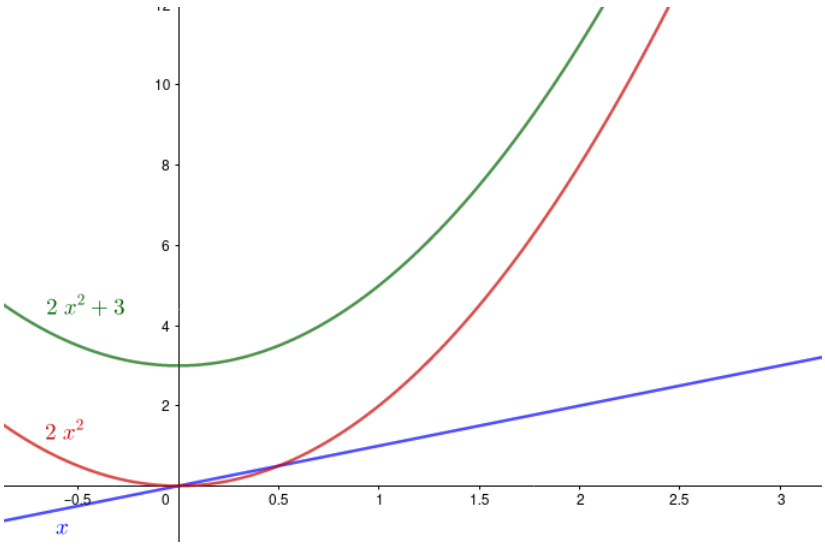
Esempi sull'Omega

- 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 $\boldsymbol{f(n)}$ **è in** $\boldsymbol{\Omega(n^m)}$

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 $n \to +\infty$):

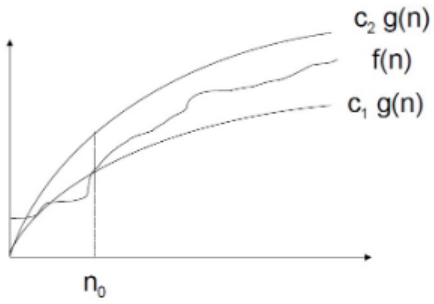
$$1 \prec log^a(n) \prec \sqrt[b]{n} \prec n^c \prec d^n \prec n! \prec n^n$$

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 $n \to +\infty$:

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

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

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

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**:

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

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                                  Θ(1)
var += 10 * 10                            Θ(1) + Θ(1) + Θ(1) = Θ(1)
print("Il valore di var è:", var)         Θ(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:

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

```
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:

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

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**:

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

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

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

Dunque, il costo finale dell'algoritmo sarà:

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

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 $if n\%2$ 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

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

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.

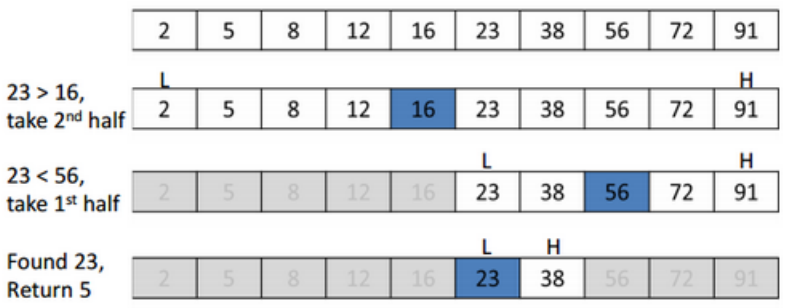
Ciò che il nostro cervello svolge (inconsciamente) è un'altra tipologia di algoritmo di ricerca, chiamata **ricerca binaria**: una volta aperto il dizionario ad una pagina casuale, abbiamo tre opzioni:

- 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 obbligatoriamente 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 $n > 1$

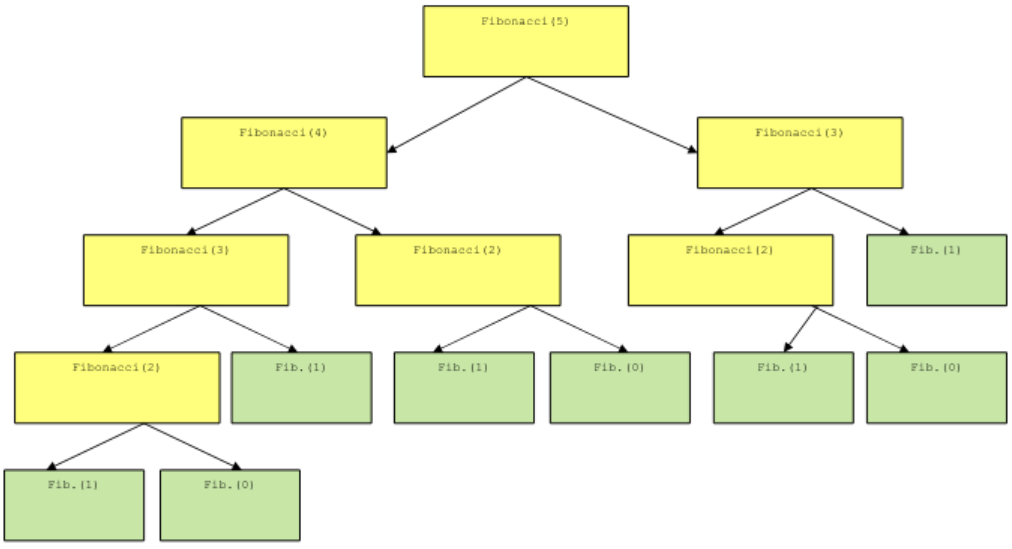
  In questo caso, viene naturale progettare l'algoritmo in termini di **ricorsione**, poiché l'n-esimo numero di Fibonacci viene calcolato tramite altri due numeri di Fibonacci.

  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**

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

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

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

Casi particolari

- Consideriamo la seguente equazione di ricorrenza relativa al calcolo dell'n-esimo numero di Fibonacci

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

Sviluppando $T(n)$ otteniamo che

$$T(n) = T(n-1) + T(n-2) + \Theta(1) = T(n-2) + 2T(n-3) + T(n-4) + 3\Theta(1) =$$

$$= T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) + 6\Theta(1) = ...$$

Notiamo come in un caso del genere **non riusciamo a generalizzare il problema,** dunque non possiamo calcolare il costo asintotico in modo analogo ai casi precedenti.

In una situazione del genere, possiamo usare le **maggiorazioni** e le **minorazioni**, cercando di calcolare il **costo O grande** e il **costo** Ω:

- Considerando $T(n) = T(n-1) + T(n-2) + \Theta(1)$, possiamo certamente dire che

$$T(n-1) + T(n-2) + \Theta(1) \leq T(n) = T(n-1) + T(n-1) + \Theta(1)$$

Dunque, poiché l'equazione di ricorrenza con cui stiamo comparando la nostra equazione originale è **minore** di quest'ultima, calcolando il **costo** Θ **della nuova equazione** otteniamo anche il **costo O grande dell'equazione iniziale**:

$$T_1 = \begin{cases} T_1(n) = T_1(n-1) + T_1(n-1) + \Theta(1) = 2T_1(n-1) + \Theta(1) \\ T_1(1) = \Theta(1) \end{cases}$$

Sviluppando $T_1(n)$ otteniamo che

$$T_1(n) = 2T_1(n-1) + \Theta(1) = 2[2T_1(n-2) + \Theta(1)] + \Theta(1) =$$
$$= 2[2[2T_1(n-3) + \Theta(1)] + \Theta(1)] + \Theta(1) = ...$$

Generalizzando l'equazione, otteniamo

$$T_1(n) = 2^k T_1(n-k) + \sum_{i=0}^{k-1} 2^i \Theta(1)$$

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

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

Dunque ne segue che

$$T_1(n) = 2^{n-1}T_1(1) + \sum_{i=0}^{n-2} 2^i\Theta(1) = \Theta(2^n) + (2^{n-1} - 1) \cdot \Theta(1) = \Theta(2^n)$$

Quindi, poiché $T(n) \leq T_1(n)$, ne segue che

$$T(n) \leq T_1(n)$$
$$T(n) \leq \Theta(2^n)$$
$$T(n) = O(2^n)$$

– Considerando $T(n) = T(n-1) + T(n-2) + \Theta(1)$, possiamo certamente dire che

$$T(n-1) + T(n-2) + \Theta(1) \geq T(n) = T(n-2) + T(n-2) + \Theta(1)$$

Dunque, poiché l'equazione di ricorrenza con cui stiamo comparando la nostra equazione originale è **maggiore** di quest'ultima, calcolando il **costo Θ della nuova equazione** otteniamo anche il **costo Ω dell'equazione iniziale**:

$$T_2 = \begin{cases} T_2(n) = T_2(n-2) + T_2(n-2) + \Theta(1) = 2T_2(n-2) + \Theta(1) \\ T_2(1) = \Theta(1) \end{cases}$$

Sviluppando $T_2(n)$ otteniamo che

$$T_2(n) = 2T_2(n-2) + \Theta(1) = 2[2T_2(n-4) + \Theta(1)] + \Theta(1) =$$
$$= 2[2[2T_2(n-6) + \Theta(1)] + \Theta(1)] + \Theta(1) = ...$$

Generalizzando l'equazione, otteniamo

$$T_2(n) = 2^k T_2(n-2k) + \sum_{i=0}^{k-1} 2^i\Theta(1)$$

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

$$n - 2k = 1 \Longrightarrow k = \frac{n-1}{2} \approx \frac{n}{2}$$

Dunque ne segue che

$$T_2(n) = 2^{\frac{n}{2}}T_2(1) + \sum_{i=0}^{\frac{n}{2}-1} 2^i\Theta(1) = \Theta(2^{\frac{n}{2}}) + (2^{\frac{n}{2}} - 1)\Theta(1) = \Theta(2^{\frac{n}{2}}) = \Theta(\sqrt{2^n})$$

Quindi, poiché $T(n) \geq T_2(n)$, ne segue che

$$T(n) \geq T_2(n)$$
$$T(n) \geq \Theta(\sqrt{2^n})$$
$$T(n) = \Omega(\sqrt{2^n})$$

Dunque, poiché il costo asintotico di **limite inferiore** e quello di **limite superiore** differiscono, **non possiamo decretare un costo asintotico stretto** per tale equazione di ricorrenza

$$T(n) = \Omega(\sqrt{2^n}), \quad T(n) = O(2^n)$$

dunque, con opportune costanti c_1 e c_2, otteniamo

$$c_1 \cdot \sqrt{2^n} \leq T(n) \leq c_2 \cdot 2^n$$

6.2 Metodo di sostituzione

L'idea alla base del **metodo di sostituzione** risulta più astratta ed "azzardata" rispetto al metodo iterativo: viene **ipotizzata una soluzione** per l'equazione di ricorrenza data e si **dimostra** tale ipotesi tramite l'**induzione**. Lo svantaggio di tale metodo risulta essere quello di poter procedere solo per **tentativi** tra varie **maggiorazioni** e **minorazioni**.

Consideriamo ancora la seguente equazione

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

1. **Ipotizziamo la soluzione** $T(n) = O(n)$, ossia che $T(n) \leq k \cdot n$ per una certa costante k indeterminata
2. **Eliminiamo la notazione asintotica** dall'equazione, sostituendo i due $\Theta(1)$ con due costanti c e d fissate

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

3. **Analizziamo il caso base**, ossia quando $n = 1$

$$T(1) \leq k \cdot 1$$

Poiché sappiamo che $T(1) = d$, otteniamo che la disuguaglianza è vera se e solo se

$$d \leq k$$

4. **Consideriamo il passo induttivo**, ossia per un n generico

$$T(n) \leq kn$$

Poiché sappiamo che $T(n) = T(n-1) + c$, otteniamo che

$$T(n-1) + c \leq kn$$

Tuttavia, per **ipotesi**, sappiamo anche che $T(n-1) \leq k(n-1)$, dunque riscriviamo la diseguaglianza come

$$k(n-1) + c \leq kn$$
$$kn - k + c \leq kn$$
$$-k + c \leq 0$$
$$c \leq k$$

Dunque la diseguaglianza è vera se e solo se $c \leq k$

5. Poiché esiste sempre un valore k tale che $k \geq c$ e $k \geq d$, la **soluzione ipotizzata** $T(n) \leq kn$ **è vera,** dunque $T(n) = O(n)$
6. **Ipotizziamo la soluzione** $T(n) = \Omega(n)$, ossia che $T(n) \geq h \cdot n$ per una certa costante h indeterminata
7. **Analizziamo il caso base**, ossia quando $n = 1$

$$T(1) \geq h \cdot 1$$

Poiché sappiamo che $T(1) = d$, otteniamo che la disuguaglianza è vera se e solo se

$$d \geq h$$

8. **Consideriamo il passo induttivo**, ossia per un n generico

$$T(n) \geq hn$$

Poiché sappiamo che $T(n) = T(n-1) + c$, otteniamo che

$$T(n-1) + c \geq hn$$

Tuttavia, per **ipotesi**, sappiamo anche che $T(n-1) \leq h(n-1)$, dunque riscriviamo la diseguaglianza come

$$h(n-1) + c \geq hn$$
$$hn - h + c \geq hn$$
$$-h + c \geq 0$$
$$c \geq h$$

Dunque la diseguaglianza è vera se e solo se $c \geq h$

9. Poiché esiste sempre un valore h tale che $h \leq c$ e $h \leq d$, la **soluzione ipotizzata** $T(n) \geq kn$ **è vera,** dunque $T(n) = \Omega(n)$
10. Poiché $T(n)$ **è sia in** $O(n)$ **sia in** $\Omega(n)$, possiamo concludere che $T(n) = \Theta(n)$

È necessario sottolineare che, in tale caso, avremmo potuto **ipotizzare e verificare** anche le soluzioni $O(n^2), O(2^n), ...$ e le soluzioni $\Omega(\sqrt{n}), \Omega(log(n)), ...$ Poiché esistono più soluzioni, è necessario fare molta attenzione alle ipotesi effettuate, dato che, ovviamente, l'obiettivo è **stimare i costi asintotici il più stretti possibile**.

6.3 Metodo dell'albero

Il **metodo dell'albero** corrisponde esattamente alla **rappresentazione grafica del metodo iterativo**, rendendolo di interpretazione più semplice rispetto alla sua controparte scritta.

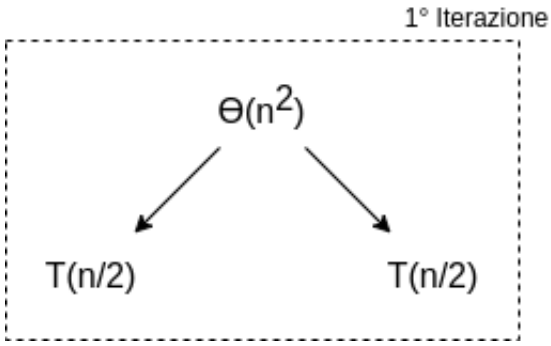
Consideriamo la seguente equazione di ricorrenza:

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

- Utilizzando il **metodo iterativo**, rappresenteremmo $T(n)$ nella forma:

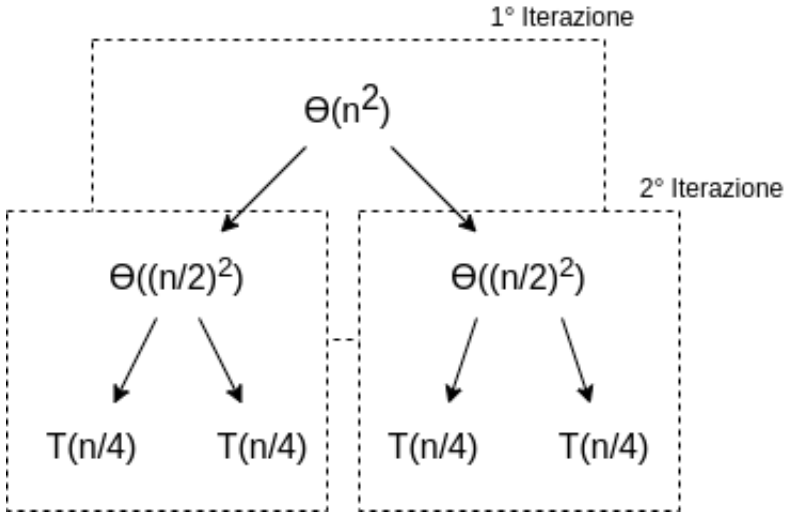
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2)$$

  Invece, la rappresentazione grafica del **metodo dell'albero** corrisponde a:



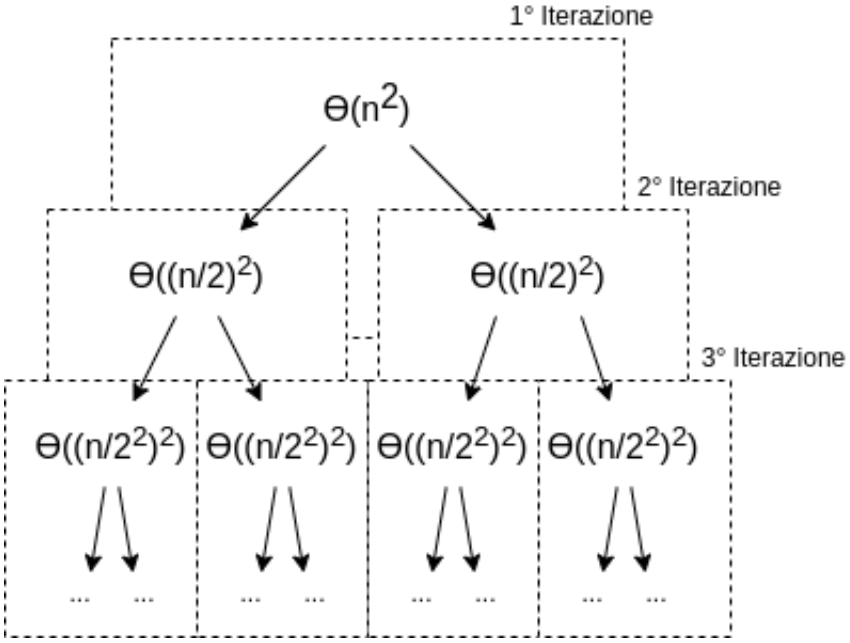
- Sviluppando la **seconda iterazione**, otteniamo

$$T(n) = 2\left(2T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right)\right) + \Theta(n^2) = 4T\left(\frac{n}{2^2}\right) + 2\Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2)$$



- Sviluppando la **terza iterazione**, otteniamo

$$T(n) = 2\left(2\left(2T\left(\frac{n}{2^3}\right) + \Theta(n^2)\right) + \Theta(n^2)\right) + \Theta(n^2) =$$
$$= 8T\left(\frac{n}{2^3}\right) + 4\Theta\left(\left(\frac{n}{2^2}\right)^2\right) + 2\Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2)$$



- A questo punto, effettuiamo il passaggio di **generalizzazione** del metodo iterativo in vista della **k-esima iterazione**

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

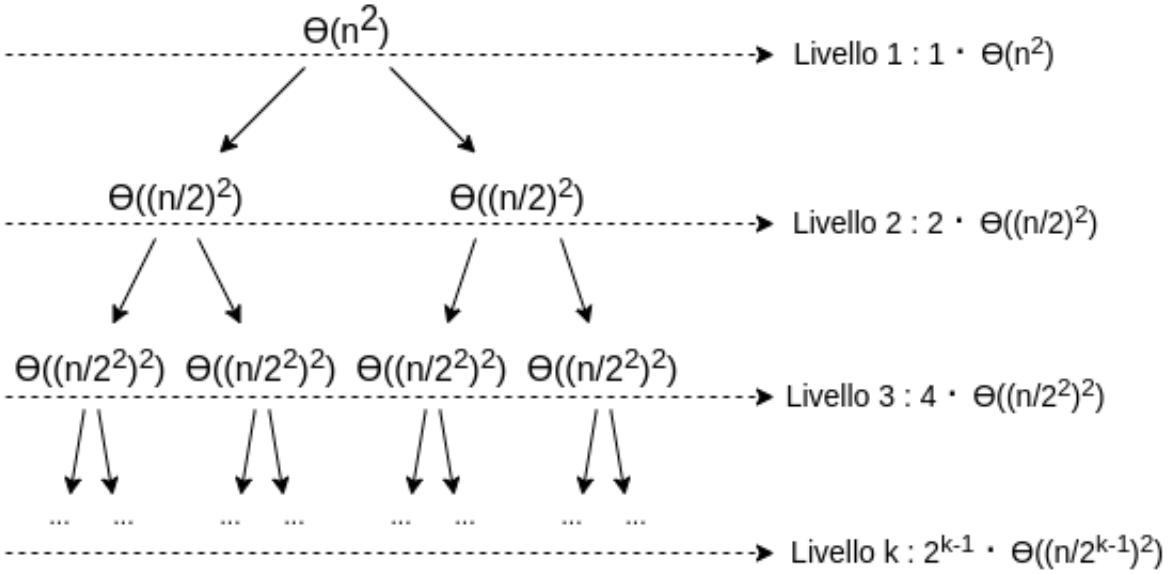
Tali iterazioni vengono eseguite finché

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

Sostituendo k otteniamo

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

- Analizziamo ora invece cosa accade ad ogni **livello dell'albero** che siamo andati a creare



- **Sommando tutti i livelli**, otteniamo che

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

- Il numero di iterazioni totali può essere trovato analogamente al metodo iterativo (ossia trovando il valore di k, che abbiamo detto essere $k = log(n)$)

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

Notiamo quindi che utilizzando entrambi i metodi abbiamo ottenuto lo stesso identico risultato tramite gli **stessi identici calcoli**. Il metodo dell'albero, quindi, si conferma essere nient'altro che una rappresentazione grafica del metodo iterativo.

6.4 Metodo principale

L'idea dietro al metodo principale consiste nell'avere una **formula** (o "ricetta") in grado di calcolare in modo estremamente rapido il costo computazionale di un determinato algoritmo. L'unico svantaggio di tale metodo, è la sua **estrema limitazione**, poiché esso **funziona solo con equazioni nel formato**

$$T(n) = \alpha \cdot \Theta\left(\frac{n}{\beta}\right) + f(n)$$

dove $T(1) = \Theta(1)$ e dove $f(n)$ può essere un qualsiasi costo Θ

L'**enunciato** del teorema del metodo principale afferma che

Theorem 3. Metodo principale

Dati $\alpha \geq 1, \beta > 1$, una funzione asintoticamente positiva $f(n)$ ed un'equazione di ricorrenza di forma

$$T = \begin{cases} T(n) = \alpha \cdot \Theta\left(\frac{n}{\beta}\right) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

abbiamo che:

- Se $f(n)$ equivale a
$$f(n) = O(n^{log_\beta(\alpha)-\varepsilon})$$
per qualche costante $\varepsilon > 0$, allora
$$T(n) = \Theta(n^{log_\beta(\alpha)})$$

- Se $f(n)$ equivale a
$$f(n) = \Theta(n^{log_\beta(\alpha)}$$
allora
$$T(n) = \Theta(n^{log_\beta(\alpha)} \cdot log(n))$$

- Se $f(n)$ equivale a
$$f(n) = \Omega(n^{log_\beta(\alpha)+\varepsilon})$$
per qualche costante $\varepsilon > 0$ e se
$$\alpha \cdot f\left(\frac{n}{\beta}\right) \leq c \cdot f(n)$$
per qualche costante $0 < c < 1$ e per n abbastanza grande, allora
$$T(n) = \Theta(f(n))$$

Traducendo il tutto in termini semplici, all'interno del teorema principale possono verificarsi **tre casi possibili** dovuti al confronto tra $f(n)$ e $n^{log_\beta(\alpha)}$:

- **Caso 1**: Se il **più grande dei due è** $n^{log_\beta(\alpha)}$, allora il costo finale è

$$T(n) = \Theta(n^{log_\beta(\alpha)})$$

- **Caso 2**: Se sono **uguali**, allora il costo finale è

$$T(n) = \Theta(f(n) \cdot log(n)) = \Theta(n^{log_\beta(\alpha)} \cdot log(n))$$

- **Caso 3**: Se il **più grande dei due è** $f(n)$, allora il costo finale è

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

ATTENZIONE: è necessario sottolineare che in questo contesto con la terminologia "più grande" si intende il valore **polinomialmente più grande**, per via della presenza del fattore ε

Esempi

Ai fini di dimostrare le applicazioni del metodo principale, in tutti i seguenti esempi verrà dato per assunto che $T(1) = \Theta(1)$

- Consideriamo la seguente equazione ricorsiva

$$T(n) = 9T\left(\frac{n}{2}\right) + \Theta(n)$$

  - $\alpha = 9$, $\beta = 2$
  - $f(n) = \Theta(n)$
  - $n^{log_\beta(\alpha)} = n^{log_3(9)} = n^2$
  - Verifichiamo facilmente che ci troviamo all'interno del **Caso 1**, poiché ponendo $\varepsilon = 1$ otteniamo

$$f(n) = O(n^{log_\beta(\alpha)-\varepsilon}) = O(n^{log_3(9)-1}) = O(n)$$

  - Dunque, il costo asintotico sarà

$$T(n) = \Theta(n^{log_\beta(\alpha)}) = \Theta(n^2)$$

- Consideriamo la seguente equazione ricorsiva

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

  - $\alpha = 1$, $\beta = \frac{3}{2}$
  - $f(n) = \Theta(1)$
  - $n^{log_\beta(\alpha)} = n^{log_{\frac{3}{2}}(1)} = n^0 = 1$
  - Verifichiamo facilmente che ci troviamo all'interno del **Caso 2**, poiché

$$f(n) = n^{log_\beta(\alpha)}$$

$$\Theta(1) = \Theta(1)$$

  - Dunque, il costo asintotico sarà

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

- Consideriamo la seguente equazione ricorsiva

$$T(n) = 3T\left(\frac{n}{4}\right) + \Theta(n \cdot log(n))$$

  - $\alpha = 3$, $\beta = 4$
  - $f(n) = \Theta(n \cdot log(n))$
  - $n^{log_\beta(\alpha)} = n^{log_4(3)} \approx n^{0.7}$
  - Verifichiamo facilmente che ci troviamo all'interno del **Caso 3**, poiché ponendo $\varepsilon = 0.1$ otteniamo

$$f(n) = \Omega(n^{log_\beta(\alpha)+\varepsilon})$$

$$f(n) = \Omega(n^{log_4(3)} + 0.1) \approx n^{0.8}$$

  - Tuttavia, ricordiamo che è necessario prima **verificare** che

$$\alpha \cdot f\left(\frac{n}{\beta}\right) \leq c \cdot f(n)$$

$$3 \cdot \frac{n}{4} \cdot log\left(\frac{n}{4}\right) \leq c \cdot n \cdot log(n)$$

$$3 \cdot \frac{n}{4} \cdot log\left(\frac{n}{4}\right) \leq c \cdot n \cdot log(n)$$

  - Notiamo facilmente come porre $c = \frac{3}{4}$ sia sufficiente a soddisfare la disequazione

$$\frac{3n}{4} \cdot log\left(\frac{n}{4}\right) \leq \frac{3n}{4} \cdot log(n)$$

$$log\left(\frac{n}{4}\right) \leq log(n)$$

  - Dunque, il costo asintotico sarà

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

- Consideriamo la seguente equazione ricorsiva

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

  - $\alpha = 2$, $\beta = 2$
  - $f(n) = \Theta(n \cdot log(n))$
  - $n^{log_\beta(\alpha)} = n^{log_2(2)} = n$
  - In questo caso, notiamo come **non rientriamo in nessuno dei tre casi**, poiché $f(n) \neq n^{log_\beta(\alpha)}$ (dunque non siamo nel caso 2) e non esiste un valore di ε che possa verificare il caso 1 o il caso 3. Di conseguenza, **non possiamo applicare il metodo principale**, richiedendo l'uso di uno degli altri tre metodi

Capitolo 7

Il Problema dell'Ordinamento

Assieme al problema della ricerca, il **problema dell'ordinamento** è in assoluto uno dei problemi più ricorrenti nell'informatica, poiché ha un'importanza fondamentale per le applicazioni di tutti i giorni, tant'è che si stima che una parte rilevante del tempo di calcolo complessivo consumato nel mondo sia relativa all'esecuzione di **algoritmi di ordinamento**.

Un **algoritmo di ordinamento** è in grado di ordinare gli elementi di un insieme sulla base di una certa **relazione d'ordine**, definita sull'insieme stesso. Per semplificare la comprensione, nelle seguenti sezioni supporremo che gli n elementi da ordinare siano dei numeri interi contenuti all'interno di un array.

Tuttavia, nei problemi reali, i dati da ordinare sono strutturati in **record**, ossia in gruppi di **informazioni** non sempre omogenee **relative allo stesso soggetto**, i quali vengono ordinati rispetto ad un'informazione specifica in grado di identificare univocamente tale record, chiamata **chiave**. Per ordinare un insieme di record, dunque, è necessario ordinarli in base alle loro chiavi.

Gli algoritmi di ordinamento che vedremo nelle sezioni successive sono:

- **Insertion Sort**
- **Selection Sort**
- **Bubble Sort**
- **Merge Sort**
- **Quicksort**
- **Heap Sort**
- **Counting Sort**

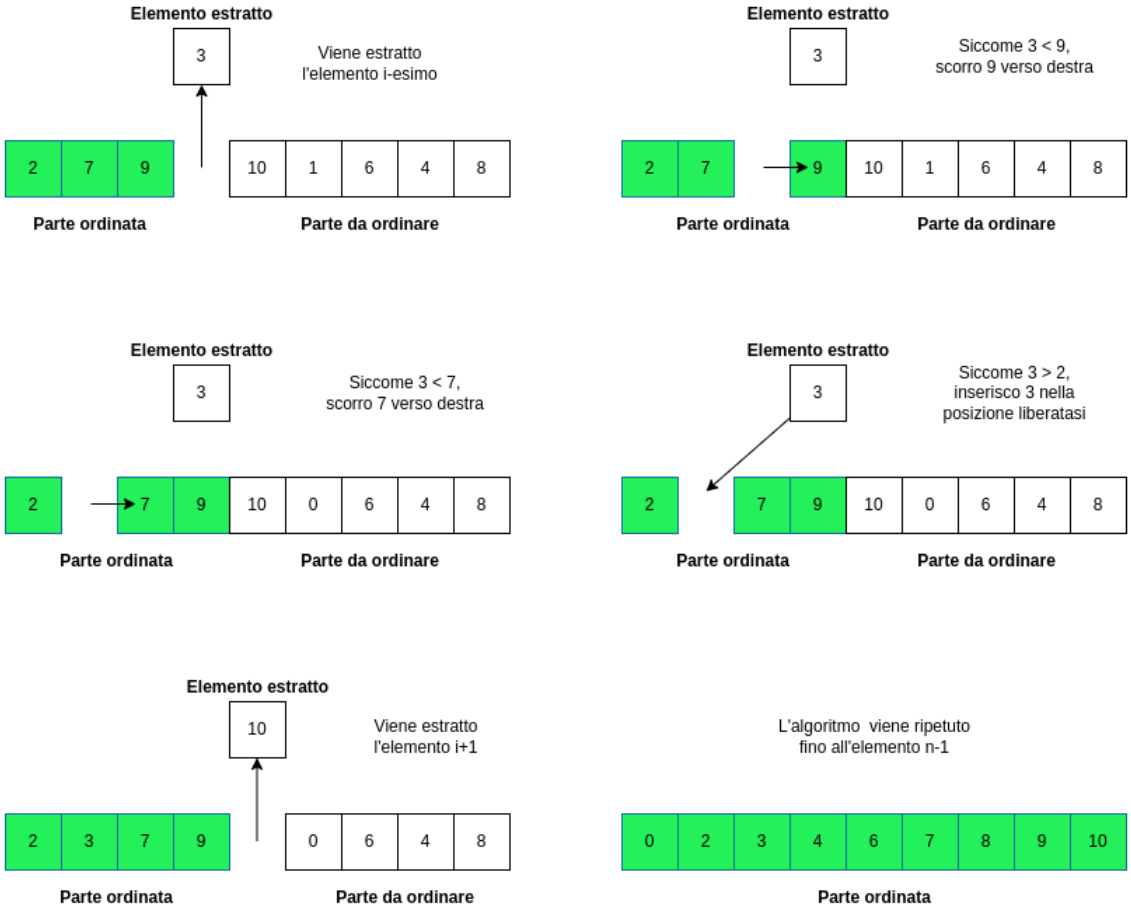
7.1 Insertion Sort

L'algoritmo di **Insertion Sort** può essere paragonato all'**ordinamento di un mazzo di carte** nella vita reale: scorrendo il mazzo analizziamo ogni carta e la mettiamo nella sua posizione corretta, inserendola in mezzo al mazzo di carte già ordinate.

In particolare, gli step dell'insertion sort sono:

- Gli elementi da ordinare sono inizialmente contenuti in un array
- Viene **estratto l'elemento della posizione** i, così da liberare la sua posizione corrente
- Tutti gli elementi alla s**ua sinistra che sono maggiori di esso** si spostano di una posizione **verso destra**, finché non viene trovato un elemento minore (o uguale) all'elemento estratto
- A quel punto, l'elemento estratto viene **inserito nella posizione liberatasi** dallo scorrimento degli altri elementi
- **INVARIANTE**: ad ogni passo i, gli elementi con indice $< i$, ossia a sinistra, sono **già ordinati**, mentre quelli con indice $> i$, ossia a destra, sono **ancora da processare**

Nota: con il termine "Invariante" si intende un predicato che, nonostante la manipolazione effettuata dall'algoritmo alla k-esima iterazione, rimane sempre vero



Lo pseudocodice corrispondente di tale algoritmo risulta quindi essere:

```
def Insertion_Sort(A)
    for j in range(1,len(A)):
        x = A[j]
        i = j - 1
        while (i>=0 and A[i]>x)
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = x
```

Il numero di iterazioni del ciclo for esterno risulta quindi essere $n-1$, mentre il numero di iterazioni del ciclo while interno si suddivide in un caso migliore ed un caso peggiore:

- **Caso migliore**: Se il primo elemento precedente a quello estratto è minore di esso (una sola operazione)
- **Caso peggiore**: Se ogni elemento precedente a quello estratto è maggiore di esso (j operazioni, dove j è l'indice dell'elemento estratto)

Dunque, nel caso in cui il **ciclo while rientri sempre nel caso migliore** (possibile solo quando la lista è già ordinata), il costo computazionale sarà

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

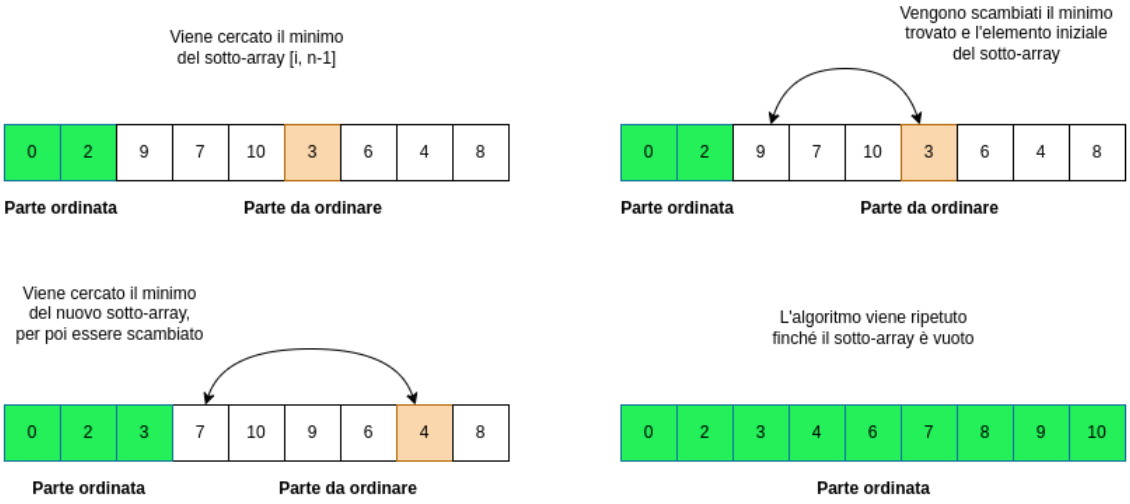
Nel caso in cui il **ciclo while rientri sempre nel caso peggiore,** invece, il costo computazionale sarà:

$$T(n) = \sum_{j=0}^{n-1} [\Theta(1) + \Theta(j)] = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

7.2 Selection Sort

Il **Selection Sort**, risulta essere il più intuitivo tra tutti gli algoritmi di ordinamento per via della sua estrema semplicità:

- Viene **ricercato il minimo** all'interno dell'array
- Il minimo trovato viene **scambiato di posizione** con il primo elemento dell'array
- Successivamente, l'algoritmo viene ripetuto sul **sotto-array di dimensione n-1** (poiché il primo elemento è già ordinato), fino a quando il sotto-array non sarà vuoto



Lo pseudocodice di tale algoritmo risulta quindi di facile implementazione:

```
def Selection_Sort(A)
    for i in range(len(A)-1):
        m = i
        for j in range(i+1,len(A)):
            if (A[j] < A[m])
                m = j
            A[m],A[i]=A[i],A[m]
```

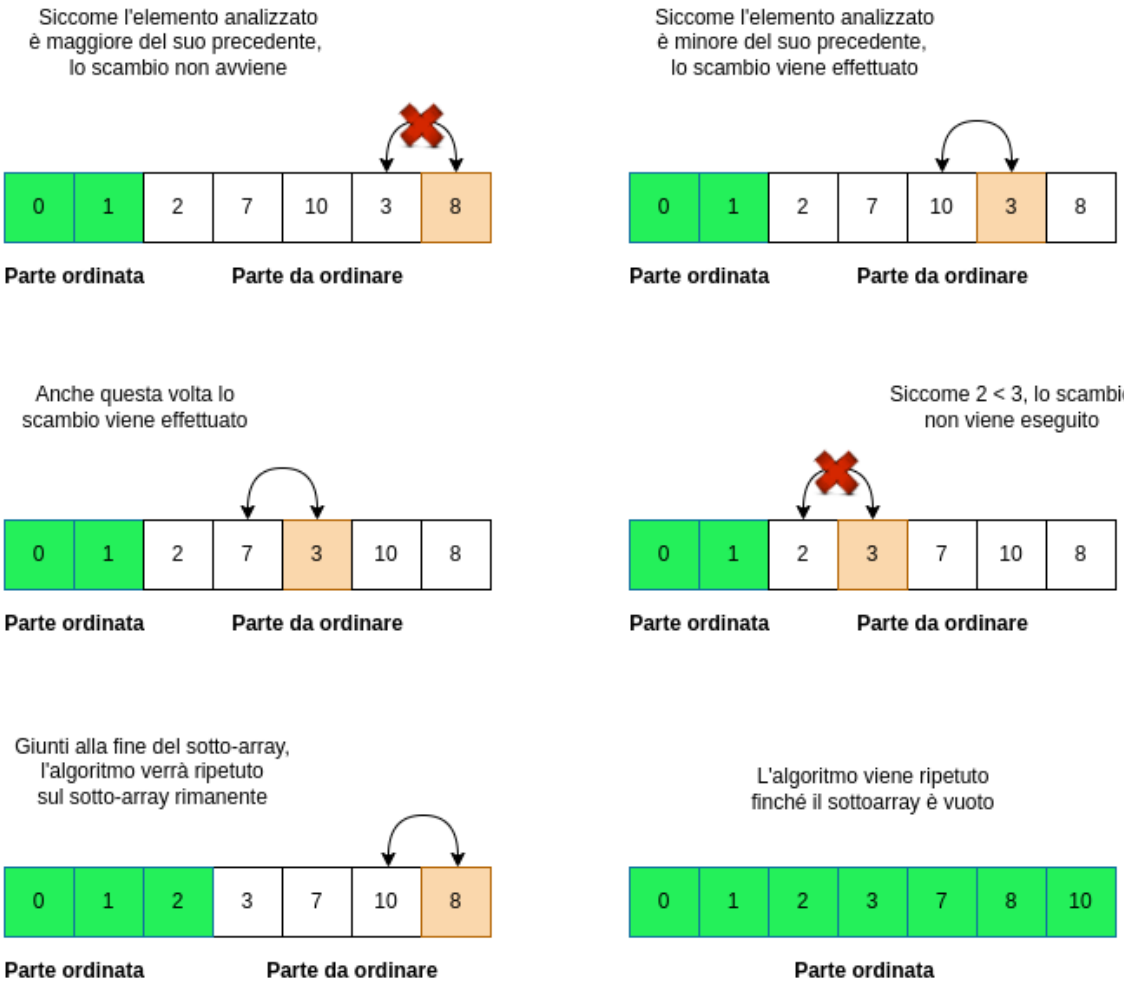
A differenza dell'Insertion Sort, tale algoritmo non possiede un caso peggiore ed un caso migliore, bensì **il suo costo corrisponde sempre** $\Theta(n^2)$:

$$T(n) = \sum_{j=0}^{n-2} [\Theta(1) + (n-j) \cdot \Theta(1) + \Theta(1)] = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

7.3 Bubble Sort

L'algoritmo di Bubble Sort è strutturato in fasi:

- Partendo **da destra verso sinistra**, vengono analizzate tutte le **coppie di elementi adiacenti**
- Se l'**elemento più a destra è minore del suo elemento precedente**, allora le loro posizioni vengono **scambiate**, altrimenti no
- Una volta comparata la **coppia più a sinistra**, l'elemento minore dell'intero array risulterà trasportato nella sua posizione finale proprio come una **bolla**
- Successivamente, l'algoritmo viene ripetuto sul sotto-array composto dagli altri elementi dell'array



L'implementazione in pseudocodice di tale algoritmo risulta essere:

```
def Bubble_Sort(A)
    for i in range(len(A)):
        for j in range(len(A)-1,i,-1):
            if (A[j] < A[j - 1])
                A[j], A[j - 1]=A[j-1],A[j]
```

Come per il Selection Sort, anche il Bubble Sort non possiede casi migliori o peggiori, bensì il **suo costo computazione risulta sempre essere** $\Theta(n^2)$:

$$T(n) = \sum_{j=0}^{n-2} [\Theta(1) + (n-j) \cdot \Theta(1) + \Theta(1)] = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

7.4 Complessità di un ordinamento

7 Dopo aver visto quelli che possono essere considerati i tre algoritmi di ordinamento più *naif* ed aver osservato che ognuno di essi abbia un costo computazionale pari a $\Theta(n^2)$, ci viene naturale chiederci quale possa essere l'algoritmo di ordinamento avente il **costo minimo possibile**.

In parole povere, quindi, vogliamo stabilire un **limite minimo di costo computazionale** al di sotto del quale **nessun algoritmo di ordinamento basato su confronti fra coppie** di elementi possa andare.

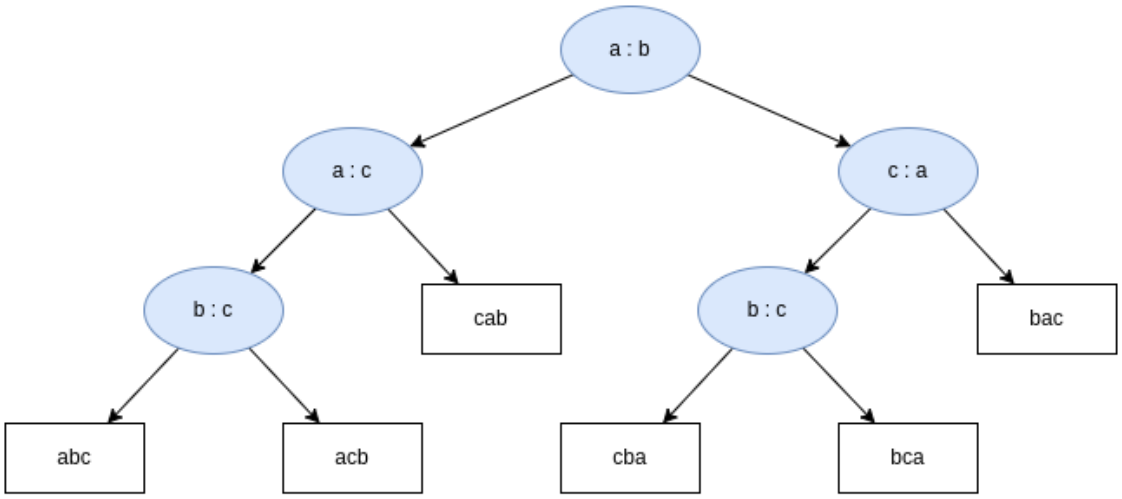
A tale scopo, risulta particolarmente utile l'**albero di decisione**, ossia uno strumento in grado di rappresentare **tutte le strade che la computazione di uno specifico algoritmo può intraprendere**, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso.

Poiché siamo in ambito di algoritmi di ordinamento basati su confronti, **ogni decisione** dell'albero corrisponderà solo a **due possibili esiti**: $a \leq b$ oppure $a > b$.

L'**albero di decisione** relativo a un qualunque algoritmo di **ordinamento basato su confronti** ha queste proprietà:

- Corrisponde ad un **albero binario** che rappresenta tutti i possibili confronti che vengono effettuati dall'algoritmo, dunque ogni **nodo** possiede due figli, mentre ogni **foglia** non ne possiede alcuno
- Ogni **nodo rappresenta un singolo confronto**, ed i due figli del nodo sono relativi ai due possibili esiti di tale confronto. Il figlio sinistro di ogni nodo corrisponderà ad un esito $a \leq b$, mentre il figlio destro corrisponderà ad un esito $a > b$.
- Ogni **foglia rappresenta una possibile soluzione** del problema, la quale è una specifica **permutazione della sequenza in ingresso**.

Ad esempio, l'albero decisionale di un algoritmo di ordinamento basato su confronti eseguito su un **array di 3 elementi** corrisponde a:



Cerchiamo quindi di determinare tale **limitazione minima del costo computazionale** dell'algoritmo:

1. Il **percorso più lungo** corrisponde al numero di confronti effettuati dall'algoritmo nel caso peggiore
2. Dato che la soluzione del problema può corrispondere ad una qualunque delle **permutazioni della sequenza in ingresso**, le **soluzioni possibili** del problema sono $n!$ preso in input un array di n elementi
3. Un **albero binario di altezza h non può contenere più di 2^h foglie**

Dalle precedenti tre osservazioni, quindi, ne segue che l'altezza h deve essere un valore tale per cui:

$$2^h \geq n! \quad \Longrightarrow \quad h \geq log(n!)$$

Come abbiamo già visto nella sezione 2.2.1, sappiamo che $log(n!) = \Theta(n\,log(n))$, dunque possiamo affermare che

$$h \geq log(n!)$$
$$h \geq \Theta(n\,log(n))$$
$$h = \Omega(n\,log(n))$$

Dunque, possiamo stipulare il seguente teorema

Theorem 4. Costo di un ordinamento con confronti

Il costo computazionale di **qualsiasi algoritmo di ordinamento basato su confronti** è $\Omega(n\,log(n))$

7.5 Merge Sort

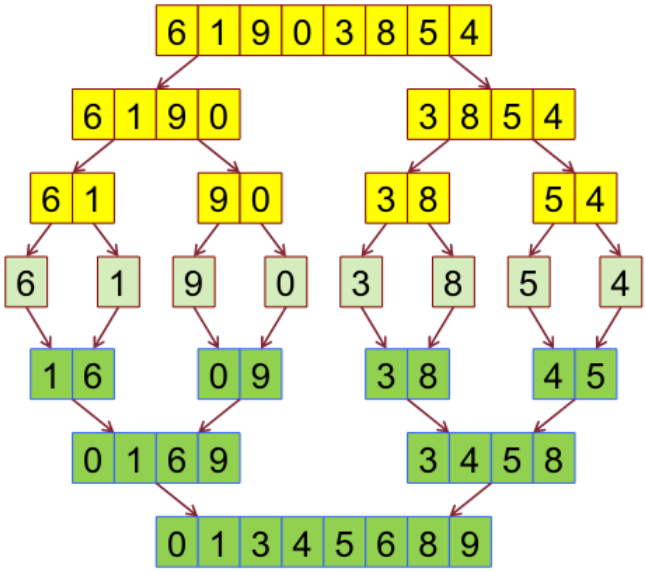
L'algoritmo **Merge Sort** è un algoritmo di ordinamento basato sull'utilizzo della **tecnica algoritmica** chiamata **divide et impera**:

- Il problema originale viene scomposto in sotto-problemi di dimensione inferiore **(divide)**
- I sotto-problemi vengono risolti richiamando ricorsivamente l'algoritmo stesso **(impera)**
- Le soluzioni dei sotto-problemi vengono utilizzate per comporre la soluzione del problema originale **(combina)**

Questa tecnica algoritmica non è fine solo a questo algoritmo, bensì essa risulta estremamente impiegata all'interno dello sviluppo di algoritmi complessi.

Nell'ambito del Merge Sort, l'approccio utilizzato è il seguente:

- **Divide**: la sequenza di n elementi viene divisa in due sotto-sequenze di $\frac{n}{2}$ elementi
- **Impera**: viene richiamato l'algoritmo ricorsivamente su entrambe le sotto-sequenze
- **Caso base**: la ricorsione termina quando la sotto-sequenza contiene un solo elemento, dunque è già ordinata
- **Combina**: le due sotto-sequenze, che sono già ordinate ricorsivamente, vengono fuse in un'unica sequenza, fino a tornare alla sequenza di dimensione n



Possiamo notare come, in base agli step svolti dall'algoritmo, il vero ordinamento avvenga all'interno dello step **combina**, dove un **sotto-algoritmo** interno al Merge Sort stesso, che chiameremo **"Fondi"** per comodità, unisce le due sotto-sequenze creandone una ordinata.

L'implementazione del Merge Sort, escludendo il sotto-algoritmo Fondi, risulta quindi essere estremamente facile poiché l'intero "lavoro sporco" viene svolto autonomamente dalla **ricorsione**:

```
def Merge_sort (A, ind_primo = 0, ind_ultimo = len(A)-1):
    if (ind_primo < ind_ultimo)
        ind_medio = (ind_primo+ind_ultimo)//2
        Merge_sort (A, ind_primo, ind_medio)
        Merge_sort (A, ind_medio + 1, ind_ultimo)
        Fondi (A, ind_primo, ind_medio, ind_ultimo)
```

Poiché ancora non sappiamo il comportamento del sotto-algoritmo Fondi, esprimeremo il costo computazionale come:

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

dove $S(n)$ è il costo computazionale di Fondi.

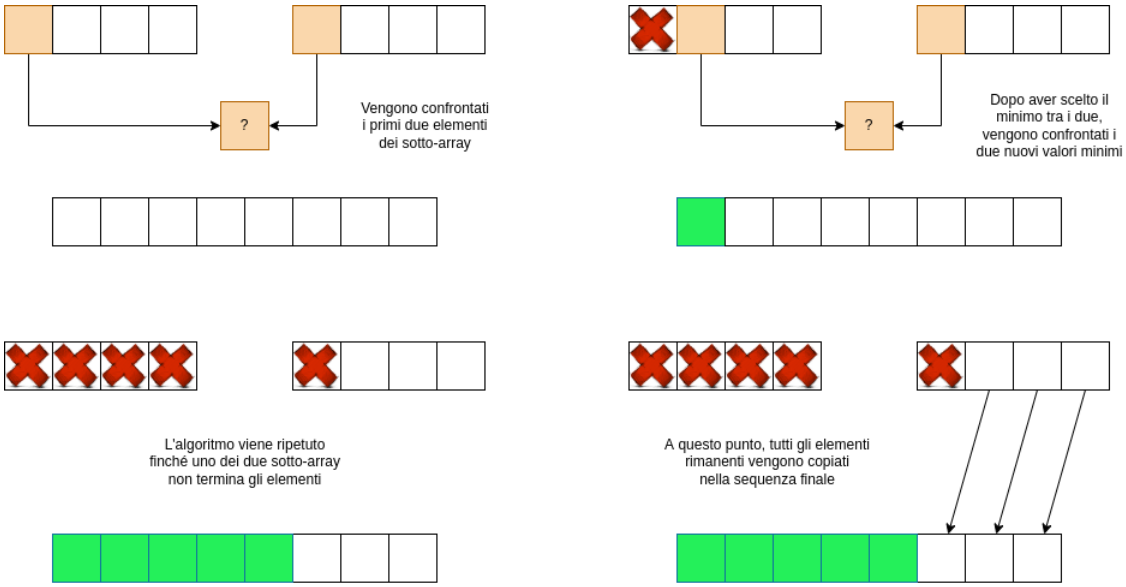
L'algoritmo Fondi

Prima di vedere il funzionamento dell'algoritmo Fondi, è necessario sottolineare che, per via della **natura stessa del Merge Sort**, è possibile utilizzare un qualsiasi algoritmo in grado di restituire una sequenza ordinata partendo dalle due sotto-sequenze. Tuttavia, ovviamente l'algoritmo Fondi utilizzato nel Merge Sort risulta essere **estremamente ottimizzato** per questo scopo, risultando in un costo computazionale ridotto al minimo.

L'algoritmo Fondi sfrutta una condizione data per assunta, ossia il fatto che **entrambe le sotto-sequenze siano ordinate**. Ovviamente, ciò risulta sempre verificato, poiché sappiamo che l'algoritmo di Merge Sort viene richiamato ricorsivamente su entrambe le sotto-sequenze ordinandole.

- Poiché le due sotto-sequenze sono ordinate, ne segue logicamente che il **valore minimo** tra il **primo elemento della prima sotto-sequenza** e il **primo elemento della seconda sotto-sequenza** sia il minimo complessivo della sequenza originale, venendo quindi spostato in quest'ultima.
- Una volta rimosso l'elemento scelto tra i due minimi, l'algoritmo verrà ripetuto, confrontando il **primo elemento della sotto-sequenza rimanente** dopo aver rimosso l'elemento precedentemente scelto e il **primo elemento dell'altra sotto-sequenza** da cui non è stato rimosso l'elemento
- Non appena una delle due sotto-sequenze ha **terminato i suoi elementi**, tutti gli elementi rimanenti nell'altra sotto-sequenza vengono direttamente **appesi uno ad uno in coda alla sequenza originale**, poiché essi sono già ordinati tra di loro

Graficamente, l'algoritmo può essere interpretato in questo modo:



Lo pseudocodice dell'algoritmo Fondi sarà quindi:

```
def Fondi (A,ind_primo,ind_medio,ind_ultimo):
    i,j = indice_primo, indice_medio+1
    B=[]

    while ((i <= ind_medio) and (j <= ind_ultimo))
        if (A[i] <= A[j]):
            B.append(A[i])
            i += 1
        else:
            B.append(A[j])
            j += 1

    //eseguito solo se il primo sotto-vettore non è terminato
    while (i <= ind_medio)
        B.append(A[i])
        i += 1

    //eseguito solo se il secondo sotto-vettore non è terminato
    while (j <= ind_ultimo)
        B.append(A[j])
        j += 1

    for i in range(len(B)):
        A[primo+i] = B[i]
```

Notiamo come il numero di iterazioni del primo ciclo possa variare da un **minimo di** $\frac{n}{2}$ (ossia il caso in cui tutti i minimi si trovino in una sola delle due sotto-sequenze) ad un **massimo di** n (ossia il caso le due sotto-sequenze terminano gli elementi in contemporanea). **Il suo costo sarà quindi** $\Theta(n)$**.**

Quanto al secondo e al terzo ciclo, invece, è necessario sottolineare che, per via delle loro condizioni, è impossibile che entrambi i cicli vengano eseguiti, dunque essi sono considerabili come un ciclo unico (poiché svolgono le stesse identiche operazioni, ma utilizzando valori diversi).

Inoltre, poiché i due cicli si occupano di trasportare tutti gli elementi rimanenti nella sotto-sequenza non terminata, il numero di iterazioni varia da **un minimo di** 1 (ossia quando solo un elemento è rimasto nella sotto-sequenza) ad **un massimo di** $\frac{n}{2}$ (coincidente con il caso migliore del primo ciclo). **Il suo costo sarà quindi** $O(n)$**.**

Infine, l'ultimo ciclo ha un **costo fisso di** $\Theta(n)$ facilmente intuibile a questo punto del corso.

Concludiamo quindi che il **costo computazionale dell'algoritmo Fondi** è:

$$S(n) = \Theta(1) + \Theta(n) + O(n) + \Theta(n) = \Theta(n)$$

Costo computazionale del Merge Sort

Una volta conosciuto il costo computazionale del sotto-algoritmo Fondi, possiamo affermare che l'equazione di ricorrenza del Merge Sort sia:

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

Utilizzando il **metodo principale** (o un qualsiasi altro metodo) otteniamo che il **costo computazionale finale del Merge Sort** corrisponde a $\Theta(n\,log(n))$, risultando quindi nel **costo più efficiente possibile** per un algoritmo di ordinamento basato su confronti (sezione 7.4).

- $\alpha = 2$
- $\beta = 2$
- $f(n) = \Theta(n)$
- $n^{log_\beta(\alpha)} = n^{log_2(2)} = n$
- Siccome $f(n) = n^{log_\beta(\alpha)}$, rientriamo nel **caso 2**. Dunque, ne segue che

$$T(n) = f(n) \cdot log(n) = \Theta(n\,log(n))$$

Osservazioni ed Ottimizzazioni del Merge Sort

L'**operazione di fusione non può essere effettuata "in loco"** (ossia aggiornando direttamente l'array su cui viene applicato senza dover utilizzare un ulteriore array di supporto) senza incorrere in un **peggioramento del costo computazionale**.

Difatti, all'interno dell'array originale bisognerebbe **fare spazio via via al minimo successivo**, ma questo costringerebbe a spostare di una posizione tutta la sotto-sequenza rimanente per ogni nuovo minimo, il che avrebbe un costo di $\Theta(n)$ per ciascun elemento da inserire, **aumentando così il costo dell'algoritmo Fondi** da $\Theta(n)$ a $\Theta(n^2)$.

A sua volta, ciò andrebbe a modificare l'equazione di ricorrenza del Merge Sort in

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

il cui costo computazionale finale corrisponde a $\Theta(n^2)$, un **netto peggioramento** rispetto al costo originale $\Theta(n\,log(n))$.

Di conseguenza, quindi, per mantenere il costo computazionale del Merge Sort al minimo possibile, è necessaria un'**elevata quantità di memoria**, poiché ad ogni richiamo dell'algoritmo Fondi è necessario creare un nuovo array (e ricordiamo che si tratta di un algoritmo ricorsivo).

Tuttavia, nonostante il Merge Sort abbia un costo di $\Theta(n\,log(n))$ mentre l'Insertion Sort abbia un costo di $O(n^2)$, è necessario puntualizzare che le costanti per i limiti asintotici sono tali che l'**Insertion Sort è più veloce del Merge Sort per valori piccoli di** n.

Dunque, possiamo ipotizzare che abbia senso **usare l'Insertion Sort all'interno del Merge Sort quando i sotto-problemi diventano sufficientemente piccoli**.

Ipotizziamo quindi il seguente algoritmo, dove k è il **limite** che stabilisce se **continuare** la catena di ricorsione o **interromperla** richiamando l'Insertion Sort:

```
def Merge_Insertion (A, k, primo, ultimo, dim):
    if dim>k:
        medio =(primo+ultimo)//2
        Merge_Insertion (A,k, primo, medio, medio-primo+1)
        Merge_Insertion (A,k,medio+1, ultimo, ultimo-primo)
        Fondi(primo, medio, ultimo)
    else:
        InsertionSort(primo, ultimo)
```

L'equazione di ricorrenza di tale algoritmo, poiché il caso base coincide con il richiamo dell'Insertion Sort, sarà quindi:

$$T(n) = \begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \\ T(k) = O(k^2) \end{cases}$$

Calcoliamo quindi il suo costo utilizzando il metodo iterativo:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = 2\left[2T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right)\right] + \Theta(n) = ...$$

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

Poiché il **caso base** viene raggiunto quando $\frac{n}{2^h} = k$, ne segue che $h = log(\frac{n}{k}$. Dunque l'equazione diventerà:

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

Se $k = O(log(n))$ (dunque $k \leq c \cdot log(n)$), otteniamo che

$$T(n) = \Theta(n\, k) + \Theta(n\, log(n)) - \Theta(n\, log(k)) =$$
$$= \Theta(n\, log(n)) + \Theta(n\, log(n)) - \Theta(n\, log(log(n))) = \Theta(n\, log(n))$$

Concludiamo quindi che se per valori $n \leq c \cdot log(n)$ viene utilizzato l'Insertion Sort internamente al Merge Sort, otteniamo un **costo computazionale invariato ma una riduzione notevole del costo in termini di memoria**, poiché l'Insertion Sort è in grado di lavorare in loco.

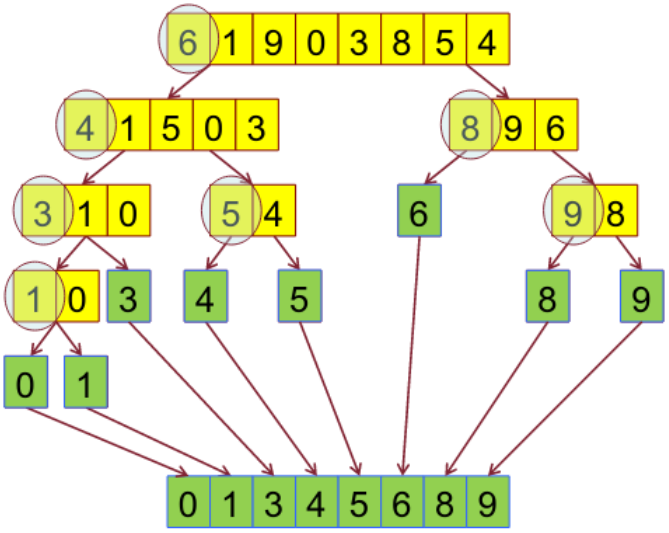
7.6 Quicksort

L'algoritmo **Quicksort** riunisce i vantaggi del Selection sort (ossia l'**ordinamento in loco**) e del Merge sort (il **ridotto tempo di esecuzione**). Tuttavia, nonostante il suo costo computazionale sia nel caso migliore sia nel caso medio sia pari a $\Theta(n\,log(n))$, il suo **svantaggio** è l'elevato costo computazionale nel suo **caso peggiore** (pari $O(n^2)$).

Come il Merge Sort, anche il Quicksort è basato sulla tecnica del **divide et impera**:

- **Divide**: nella sequenza di n elementi viene selezionato un valore come **pivot** dell'algoritmo. Solitamente, il pivot scelto corrisponde a il primo, all'ultimo o al valore a metà dell'array su cui si applica l'algoritmo.
- La sequenza viene quindi divisa in due sotto-sequenze, la prima contenente tutti gli elementi minori del pivot e la seconda contenente tutti gli elementi maggiori o uguali al pivot
- **Impera**: le due sotto-sequenze vengono ordinate richiamando ricorsivamente l'algoritmo su di esse
- **Caso base**: la ricorsione termina quando la sotto-sequenza contiene un solo elemento, dunque è già ordinata

Notiamo come, a differenza del Merge Sort, le sotto-sequenze non vengono fuse tra di loro. Difatti, il vero e proprio ordinamento viene svolto nella **fase di continua divisione ordinata ricorsiva** effettuata dall'algoritmo, ossia il **Partizionamento**.



Lo pseudocodice del Quicksort corrisponde a:

```
def Quick_sort (A, ind_primo = 0, ind_ultimo = len(A)-1)
    if (ind_primo < ind_ultimo):
        ind_medio = Partiziona(A,ind_primo,ind_ultimo)
        Quick_sort (A,ind_primo,ind_medio)
        Quick_sort (A,ind_medio+1,ind_ultimo)
```

Come per il Merge Sort, quindi, gran parte del "lavoro sporco" viene svolto da un sotto-algoritmo. A differenza dell'algoritmo Fondi, tuttavia, l'implementazione dell'**algoritmo Partiziona** non risulta particolarmente analoga alla sua interpretazione concettuale descritta precedentemente, per via della necessità stretta di dover svolgere le **operazioni in loco** e non su un array di appoggio:

```
def Partiziona(A, ind_primo, ind_ultimo):
    i, j = ind_primo - 1, ind_ultimo + 1
    pivot = A[ind_primo]

    while True:

        i += 1
        while A[i] < pivot:
            i += 1

        j -= 1
        while A[j] > pivot:
            j -= 1

        if i < j:
            A[i], A[j] = A[j], A[i]     #inverti A[i] e A[j]
        else:
            return j
```

Nella pratica, una volta considerata la *"magia della ricorsione"*, il comportamento di tale algoritmo corrisponde con il suo modello concettuale.

Per calcolare il **costo computazionale** di tale algoritmo, è necessario valutare il costo del **ciclo while esterno**, il quale è strettamente dipendete da ciò che avviene al suo interno: notiamo come ciascuno dei due cicli repeat-untill vada ad **avvicinare l'uno all'altro i due indici** i **e** j, finché essi non si **incrociano**.

Ipotizzando che l'indice i effettui k scorrimenti, ne segue quindi che j vada effettuare $n-k$ scorrimenti. La somma dei due, quindi, corrisponde ad **un totale di** n **operazioni**, dunque ad un costo complessivo di $\Theta(n)$. Il ciclo while esterno, quindi, sarà eseguito un massimo di n iterazioni, risultando in un costo complessivo dell'algoritmo pari a:

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

Quanto al costo computazionale del Quicksort, invece, è necessario considerare che l'indice restituito dall'algoritmo Partiziona vada a **suddividere la sequenza originale in due sotto-sequenze**, dove la prima conterrà k **elementi**, mentre la seconda conterrà $n-k$ **elementi**. L'equazione di ricorrenza sarà quindi:

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

Poiché non possiamo calcolare un costo strettamente asintotico di tale equazione tramite alcun metodo, possiamo ipotizzare il suo **caso peggiore**, **caso migliore** e **caso medio**:

- **Caso migliore**: corrisponde al caso in cui ad ogni ricorsione ogni sotto-array venga diviso in esattamente due metà, ossia quando $k = n - k = \frac{n}{2}$:

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

  Poiché sappiamo che $P(n) = \Theta(n)$, ne segue quindi che l'equazione di ricorrenza del caso migliore coincida con quella del Merge Sort, che sappiamo già essere $\Theta(n\,log(n))$

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

- **Caso peggiore**: corrisponde al caso in cui ad ogni ricorsione ogni sotto-array venga diviso in un sotto-array contente 1 elemento ed uno contente $n-1$ elementi, ossia quando $k = 1$:

  $$T(n) = \Theta(1) + P(n) + T(k) + T(n-k) = \Theta(1) + P(n) + T(1) + T(n-1)$$

  Sapendo che $T(1) = \Theta(1)$ e che $P(n) = \Theta(n)$, ne segue che l'equazione di ricorrenza del caso peggiore sia:

  $$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

  il cui costo è facilmente calcolabile utilizzando il **metodo iterativo**.

- **Caso medio**: ipotizziamo che il valore del pivot suddivida con **uguale probabilità**, pari a $\frac{1}{n-1}$, la sequenza da ordinare in due sotto-sequenze di dimensioni k ed $n-k$, per **tutti i valori di** k **tra** 1 **ed** $n-1$.

  Ne segue, quindi, che l'equazione di ricorrenza sia

  $$T(n) = \frac{1}{n-1}\left[\sum_{k=0}^{n-1}(T(k) - T(n-k))\right] + P(n)$$

  il cui **costo computazionale** è riconducibile, tramite il **metodo di sostituzione**, a $O(n\,log(n))$ (per questioni di praticità verrà omessa tale dimostrazione, poiché contenente calcoli molto complessi, lunghi e futili da seguire).

  Dunque, poiché sappiamo che tale equazione è $O(n\,log(n))$ e che il **teorema della complessità di un algoritmo di ordinamento basato sui confronti** impone che esso sia anche $\Omega(n\,log(n))$, possiamo concludere che il caso medio sia $\Theta(n\,log(n))$.

Osservazioni e Ottimizzazioni del Quicksort

Concludiamo quindi che il costo computazionale dell'**algoritmo di Quicksort** sia quasi sempre pari a $\Theta(n\,log(n))$, risultando quindi l'**algoritmo ideale per input di grandi dimensioni**, poiché a differenza del Merge Sort esso svolga le operazioni in loco.

A volte, però, l'**ipotesi di equiprobabilità non è soddisfatta** (ad esempio quando i valori in input sono **"poco disordinati"**), riducendo notevolmente le prestazioni dell'algoritmo. Difatti, il **caso peggiore** viene raggiunto proprio quando la **lista è già ordinata** e il **pivot** selezionato corrisponde al **primo elemento dell'array**, poiché ovviamente esso sarebbe minore di tutti i valori a suo seguito, andando ogni volta a dividere l'array in due sotto-array di lunghezza 1 e $n-1$, risultando quindi in un costo pari a $\Theta(n^2)$.

Per ovviare a tale inconveniente si possono adottare delle **tecniche volte a randomizzare la sequenza da ordinare**, in modo da eliminarne l'eventuale regolarità interna. Tali tecniche mirano a rendere l'algoritmo indipendente dall'input, consentendo di ricadere nel caso medio:

- Prima di avviare l'algoritmo d'ordinamento, **la sequenza viene randomizzata**, evitando che l'array iniziale sia parzialmente ordinato
- Durante il partizionamento **viene scelto un pivot casuale** all'interno della sequenza e non sistematicamente il valore più a sinistra, più a destra o nel mezzo

Materiale aggiuntivo

Poiché l'algoritmo di partizionamento può risultare inizialmente complesso, **è consigliata la visione di questi due video** contenenti una spiegazione passo passo dell'algoritmo:

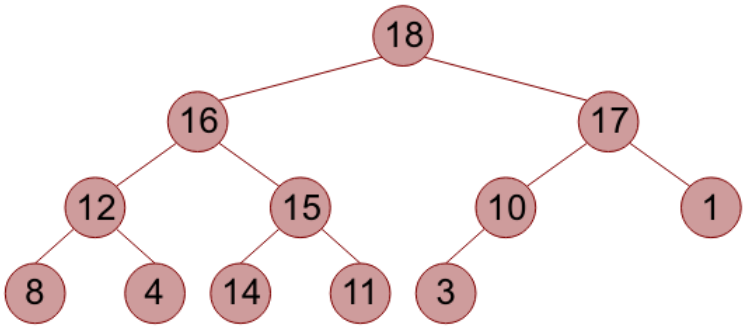
- Quicksort: Partitioning an array (KC Ang)
- Sorts 8 Quick Sort (RobEdwards)

7.7 Heap Sort

L'algoritmo **Heap Sort**, a differenza del Merge Sort e del Quicksort, sfrutta una **opportuna organizzazione dei dati**, ossia una **struttura dati**, che garantisce una o più specifiche proprietà, il cui mantenimento è essenziale per il corretto funzionamento dell'algoritmo.

7.7.1 La struttura dati Heap

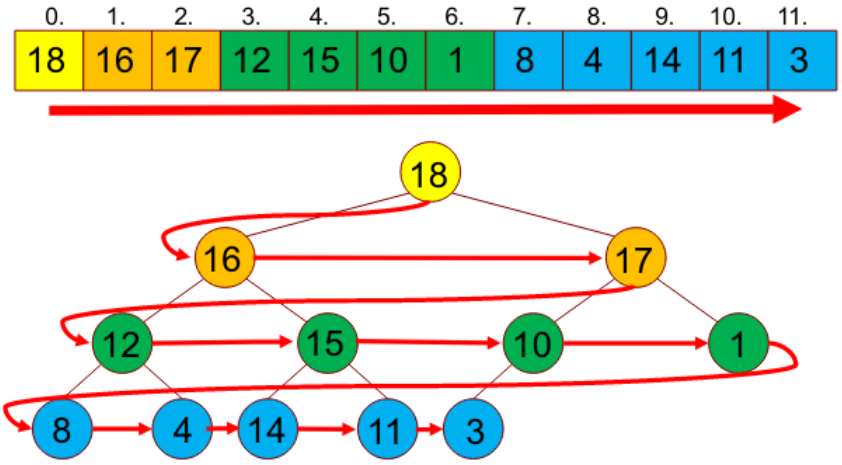
Tale struttura dati viene detta **Heap**, ossia un **albero binario** completo o quasi completo, in cui tutti i livelli, eccetto l'ultimo, sono pieni e in cui i nodi sono addensati a sinistra. La proprietà fondamentale che mette in relazione tali nodi è l'**ordinamento verticale**: la **chiave di ogni nodo** corrisponde ad un **valore maggiore o uguale** alle **chiavi dei due figli** di ogni nodo.



Nonostante la sua struttura ad albero, un Heap viene comunque **implementato tramite un array**, i cui indici vanno da 0 fino al numero di nodi dell'Heap, ossia l'**Heap Size**, decrementato di 1. Tuttavia, tale array, affinché possa essere considerato una **traslazione dell'Heap dal modello teorico al modello pratico**, dovrà rispettare alcune caratteristiche:

- L'array è **riempito a partire da sinistra** e, se contiene più elementi del valore indicato dall'Heap Size, allora gli elementi di indice $\geq heap\_size$ non fanno parte dell'Heap
- Ogni nodo dell'albero binario corrisponde a uno e un solo elemento dell'array
- La **radice dell'albero** corrisponde ad $A[0]$ e, poiché ogni chiave padre deve essere maggiore o uguale delle chiavi figlie, corrisponde al **valore massimo dell'Heap** e, di conseguenza, può essere trovato in tempo $\Theta(1)$
- Considerato il **nodo generico** $A[i]$, il suo **figlio sinistro**, se esiste, corrisponde all'**elemento** $A[2i+1]$, mentre il suo **figlio destro** corrisponde all'**elemento** $A[2i+2]$
- Di conseguenza, considerato il **nodo generico** $A[i]$, il suo **padre** sarà l'**elemento** $A\lfloor(i-1)/2\rfloor$
- Poiché **ogni livello dell'Heap contiene** 2^h **nodi**, ne segue che l'**altezza** sia $h = log(n)$, dove n è il numero di nodi

Paragonando il modello teorico al modello pratico dell'Heap, quindi, otteniamo che:



Per poter utilizzare l'**algoritmo di Heap Sort** su un vettore, quindi, è prima **necessario modificare tale vettore** in modo che vada a rispettare le **proprietà della struttura dati Heap**. In particolare, ciò viene realizzato utilizzando due funzioni ausiliarie:

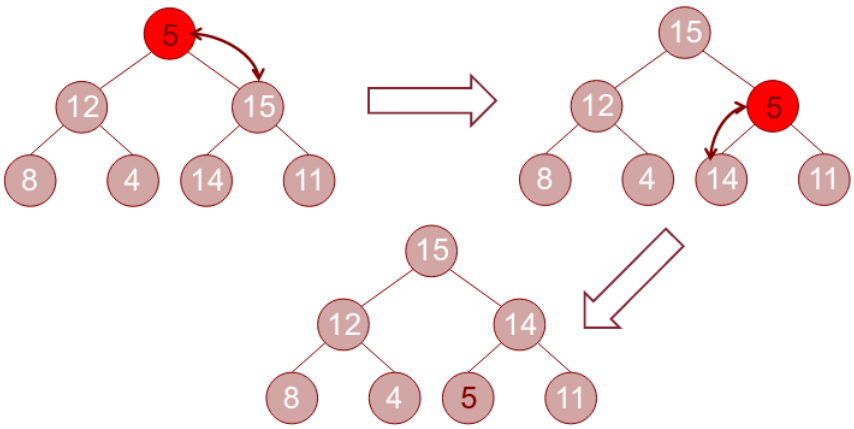
- La funzione **Heapify**
- La funzione **Buildheap**

La funzione Heapify

La **funzione Heapify** ha lo scopo di **mantenere la proprietà di Heap**, dando per assunta l'**ipotesi** che nell'albero su cui viene fatta lavorare **sia garantita la proprietà di heap per entrambi i sotto-alberi** (sinistro e destro) della radice.

Di conseguenza l'unico nodo che può **violare la proprietà di Heap** è la radice dell'albero, che può essere minore di uno o di entrambi i figli.

La funzione, quindi, **opera sulla radice confrontandola coi suoi figli** e, se necessario, la **scambia col maggiore** di suoi figli. Dopo lo scambio, viene verificato se la violazione si sia ***"trasferita"*** **sul figlio scambiato** e, se necessario, si ripete **ricorsivamente** l'operazione su tale nodo.



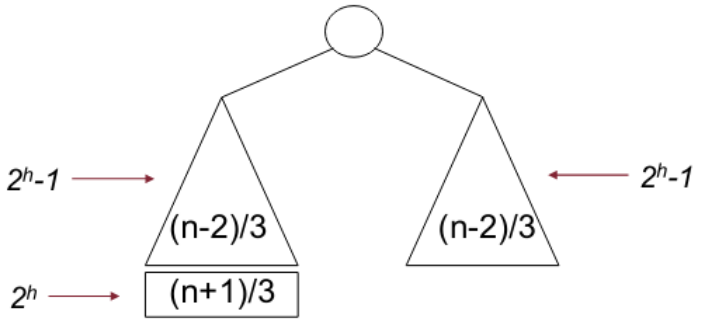
L'implementazione in pseudocodice della **funzione Heapify** corrisponde a:

```
def Heapify (A, i, heap_size)
    L = A[2i+1]
    R = A[2i+2]
    indice_max = i

    if (L < heap_size) and (A[L] > A[i]):
        indice_max=L
    if (R <= heap_size) and (A[R] > A[indice_max]):
        indice_max=R

    //se viene violata la proprietà
    if (indice_max != i)
        A[i], A[indice_max] = A[indice_max], A[i]
        Heapify (A, indice_max, heap\_size)
```

Notiamo come tutte le operazioni svolte all'interno della funzione corrispondano ad un $\Theta(1)$, fatta eccezione per la chiamata ricorsiva. Dunque, l'equazione di ricorrenza della funzione sarà

$$T(n) = \begin{cases} T(n) = \Theta(1) + T(n') \\ T(1) = \Theta(1) \end{cases}$$

dove n' è il **numero di nodi del sotto-albero più grande** tra figlio destro e figlio sinistro. Poiché ogni livello contiene 2^h nodi, tale numero **non può essere più grande di** $\frac{2}{3}n$, situazione che accade quando l'ultimo livello è pieno esattamente a metà:



Dunque, l'equazione di ricorrenza diventa:

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

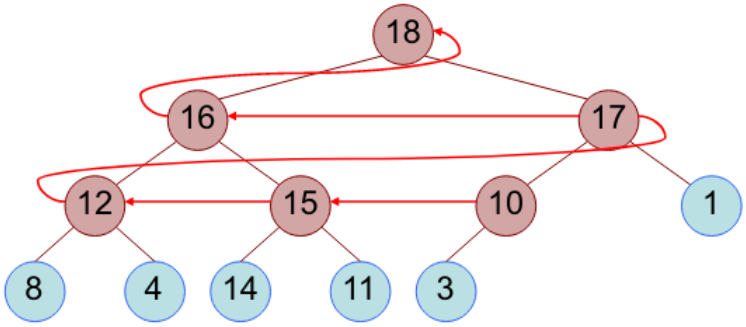
- $\alpha = 1$, $\beta = \frac{3}{2}$
- $f(n) = \Theta(1)$
- $n^{log_\beta(\alpha)} = n^{log_{\frac{3}{2}}(1)} = n^0 = 1$
- Siamo nel **caso 2**, dunque $T(n) = f(n) \cdot log(n) = \Theta(log(n))$

La funzione Buildheap

La **funzione Buildheap** si occupa di **trasformare qualunque vettore** contenente n elementi **in un Heap**, chiamando ripetutamente la **funzione Heapify** sugli opportuni nodi dello Heap.

Tuttavia, poiché la funzione Heapify dà per assunto che entrambi i sotto-alberi della radice siano a loro volta degli Heap, essa deve essere **richiamata** scorrendo l'albero per livelli **dal basso verso l'alto**.

Inoltre, poiché **ogni foglia è già un Heap** (dato che non ha figli), possiamo **ridurre il numero di chiamate a** $\frac{n}{2}$, poiché, per via delle proprietà dell'Heap, la seconda metà dell'array sarà sempre occupata solo e soltanto dalle foglie.



Lo pseudocodice della funzione Buildheap corrisponde a

```
def Build_heap (A):
    for i in revered(range(len(A)//2)):
        Heapify (A, i,heap_size)
```

il cui **costo computazionale** è pari a

$$T(n) = \frac{n}{2} \cdot \Theta(log(n)) = \Theta(n\, log(n))$$

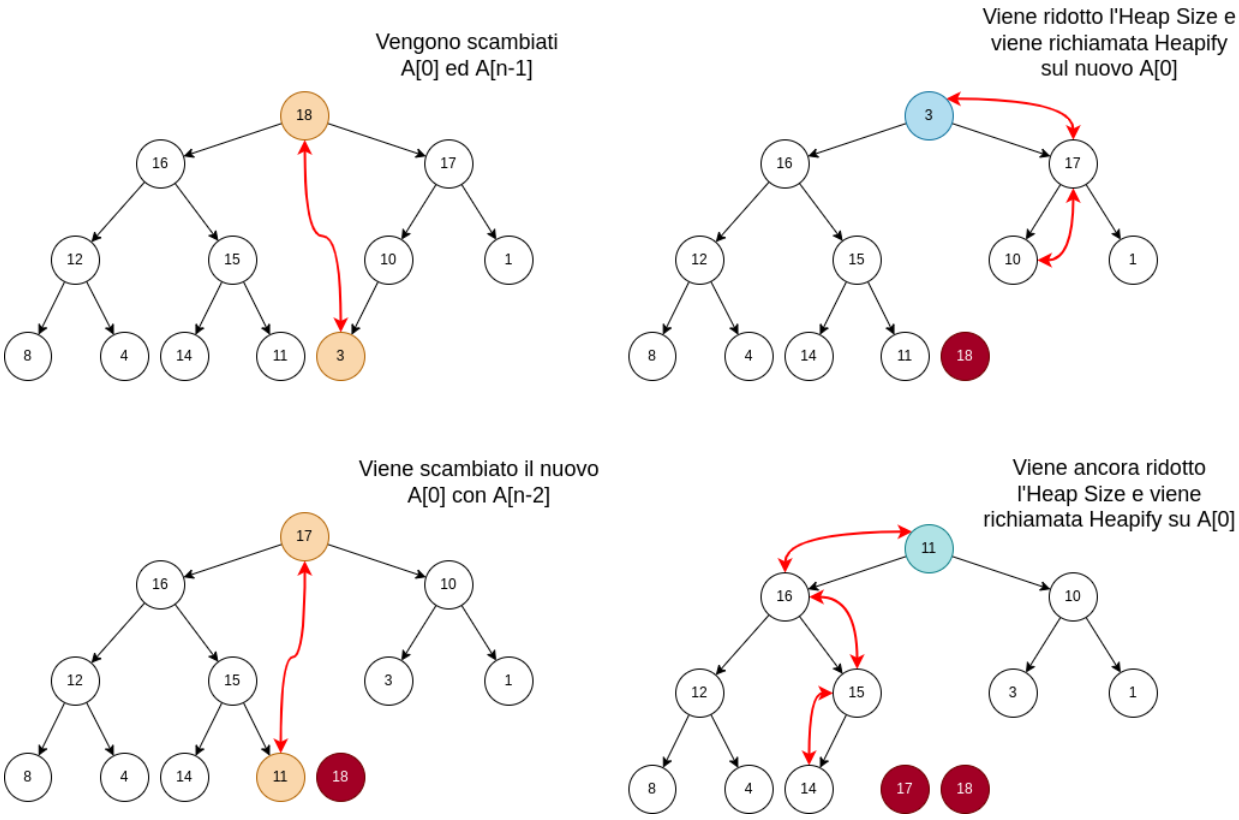
Con un **calcolo più accurato del costo**, considerando le varie proprietà dell'Heap, è possibile ricondurre il **costo computazionale a** $\Theta(n)$. Tuttavia, tale dimostrazione verrà omessa, poiché, come vedremo in seguito, la differenza tra il costo immediato $\Theta(n\, log(n))$ e il costo accurato $\Theta(n)$ **non influisce sul costo finale dell'algoritmo di Heap Sort**.

L'algoritmo di Heap Sort

Una volta comprese le **proprietà** della struttura dati Heap ed aver definito due **funzioni ausiliarie** che ci permettono di manipolarlo, andiamo a vedere il funzionamento dell'**algoritmo di ordinamento Heap Sort**:

- Viene richiamata la **funzione Buildheap** sul vettore dato in input
- Poiché il **massimo dell'Heap è in posizione** $A[0]$, per metterlo nella corretta posizione è necessario **scambiarlo con** $A[n-1]$, ossia l'ultimo elemento

- Di seguito, la **dimensione dell'Heap viene ridotta ad** $n-1$, poiché possiamo **ignorare l'ultimo elemento** dato che esso si trova già nella posizione finale.
- Poiché i due sotto-alberi della radice sono ancora degli Heap, viene richiamata la **funzione Heapify sulla radice**, che ricordiamo non essere più il massimo dell'Heap per via dello scambio, in modo da ripristinare a pieno le **proprietà dell'Heap** di dimensione ridotta
- Successivamente, il procedimento verrà **riapplicato sul nuovo massimo dell'Heap**, andando man mano a ridurre l'Heap Size fino al minimo.



Lo pseudocodice dell'Heap Sort corrisponde a:

```
def Heapsort (A):
    Build_heap(A)
    for x in reversed(range(1,len(A))):
        A[0],A[x]=A[x],A[0]
        Heapify(A, 0, x)
```

Notiamo come il ciclo for venga eseguito un totale di $n-1$ volte. Dunque, per via del richiamo della f**unzione Heapify** al suo interno (che sappiamo essere $\Theta(log(n))$), il suo costo sarà pari a $\Theta(n\,log(n))$.

A questo punto, il costo della **funzione Buildheap**, che ricordiamo essere $\Theta(n)$ con un calcolo accurato, non andrà ad influenzare il **costo finale dell'algoritmo di Heap Sort**, che sarà pari a:

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

7.8 Ordinamenti lineari

7.8.1 Counting sort

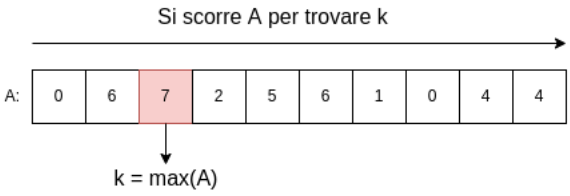
Abbiamo già visto il teorema secondo cui **ogni algoritmo di ordinamento che opera per confronti ha un costo computazionale di** $\Omega(nlog(n))$ (sezione 7.4). Vediamo ora quindi un algoritmo non basato sui confronti e che, di conseguenza, "sfugge" al limite inferiore imposto da tale teorema, riuscendo ad arrivare ad un **costo computazionale lineare** $\Theta(n)$.

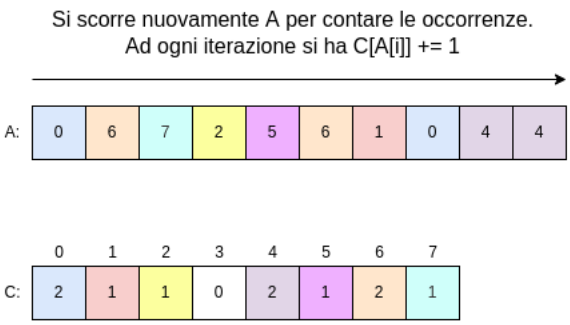
Ipotizziamo di avere un array di numeri interi da ordinare. Ciascuno degli n elementi da ordinare è **un intero di valore compreso in un intervallo** $[0..k]$, dove k corrisponde al **valore massimo dell'array**.

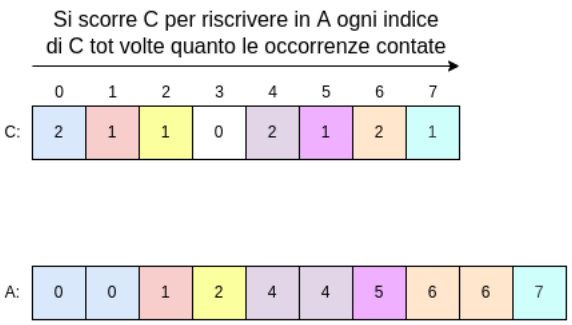
A questo punto, possiamo utilizzare **un secondo array di supporto**, i cui valori corrispondono esattamente al **numero di occorrenze dell'indice stesso**.

Infine, verrà usato il numero di occorrenze contate per riscrivere nell'array iniziale i valori **in ordine di indice**.

- Viene trovato il **valore massimo** k dell'array A
- Viene creato un **secondo array** C di larghezza $k + 1$
- Viene **conteggiato** ogni elemento di A, **incrementando di uno il corrispettivo indice nell'array** C. Esempio: se $A[i] = 6$, allora $C[6]$ viene incrementato di uno.
- Viene sovrascritto l'array A scorrendo C, ricopiando in A **ciascun indice di** C **tante volte quanto è il valore in** C **di quell'indice**.







Lo pseudocodice di tale algoritmo corrisponde a:

```
def counting_sort (A):

    k = max(A)                    #Θ(n)

    n = len(A)

    C = [0]*(k+1)                 #Θ(k)

    for j in range(n):            # n volte
        C[A[j]] +=1
        #C[i] ora contiene il numero di elementi uguali a i

    j = 0
    for i in range(k+1):            # k volte
        while (C[i] > 0)          # C[i] volte
            A[j]=i
            j+=1
            C[i]-=1
```

A questo punto, è necessario puntualizzare che:

- Il costo computazionale della funzione $max()$ è $\Theta(n)$ poiché scorriamo l'array A
- La **somma di tutti i valori contenuti in** C, una volta finito il conteggio, corrisponde a n

$$\sum_{i=0}^{k} C[i] = n$$

A questo punto, siamo in grado di scrivere l'equazione descrivente il **costo dell'algoritmo**:

$$T(n) = \Theta(n) + \Theta(n) + \Theta(1) \cdot \sum_{i=0}^{k} C[i] = \Theta(n) + \Theta(k) = \Theta(n+k)$$

Notiamo quindi che il costo computazionale si divide in **due casi**:

- Se $k \leq n$, allora $T(n) = \Theta(n)$
- Se $k > n$, allora $T(n) = \Theta(k)$

Questa **versione del Counting Sort**, tuttavia, è **adeguata solamente se non vi sono dati satellite**. Infatti, il ciclo che ricopia i valori contati in C nel vettore A di fatto ne **sovrascrive i dati**.

7.8.2 Counting Sort con Dati Satellite

Se ci dovessero essere **dati satellite**, oltre ai vettori A e C si deve introdurre un **nuovo vettore B di n elementi**, che alla fine conterrà la sequenza ordinata:

- Dopo aver conteggiato gli elementi in C, si fa una **seconda passata su C**, da sinistra a destra, partendo da $C[2]$, nella quale a **ogni elemento $C[i]$ si somma il precedente**
- Alla fine di tale fase si ha che:
  - $C[i]$ indica la **posizione corretta**, nel vettore ordinato, per l'elemento di valore pari a i più a destra fra quelli contenuti nel vettore A
  - $C[i] - C[i-1]$ indica **quanti elementi ci sono con valore pari a $C[i]$**
- Si scorre quindi il vettore A da destra a sinistra e:
  - Si **copia in B ogni elemento** $A[j] = k$ nella posizione giusta che è $C[k]$
  - Si **decrementa** di 1 il valore $C[k]$.

Lo pseudocodice di questa **seconda versione** corrisponde a:

```
def counting_sort_2 (A):
    k = max(A)
    n = len(A)
    C = [0]*(k+1)
    B = [0]*n

    for j in range(n)
        C[A[j]]+=1
        #in C[i] ora c'è il num. di elem. = i

    for i in range(1,k)
        C[i]+=C[i-1]
        #in C[i] ora c'è il num. di elem. <= i

    for j in range(n,-1)
        B[C[A[j]]]=A[j]]
        C[A[j]]-=1
    return B
```

Nonostante le modifiche rispetto alla versione precedente, il **costo computazionale rimane invariato**:

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

Capitolo 8

Strutture Dati

In un linguaggio di programmazione, un **dato** è un valore che una variabile può assumere, mentre il **tipo di un dato** rappresenta una collezione di valori e un insieme di operazioni ammesse su questi valori (intero, carattere, stringa, ...)

Le **strutture dati**, invece, sono particolari tipi di dato, caratterizzate dall'**organizzazione dei dati**, più che da tipo di dato stesso. (lista, pila, coda, albero, ...)

Le strutture dati, quindi, sono:

- Un **modo sistematico di organizzare i dati**
- Un **insieme di operatori** che permettono di manipolare la struttura
- In grado di **memorizzare e manipolare insiemi dinamici** (composti solitamente da una **chiave**, ossia un valore in grado di identificare l'insieme in modo univoco, ed da **dati satelliti**, ossia qualsiasi altro dato legato all'insieme stesso), i cui elementi possono variare nel tempo.
- **Lineari o non lineari** (a seconda che esista o no una sequenzializzazione dei valori)
- **Statiche o dinamiche** (a seconda che possano variare la dimensione nel tempo)
- **Omogenee o disomogenee** (rispetto ai tipi di dati contenuti)

Le **operazioni** che vengono svolte su ogni struttura dati possono essere racchiuse in **due categorie**:

- **Operazioni di interrogazione**:
  - **Search(S, k)**: recuperare l'elemento con chiave di valore k, se è presente in S, restituire un valore speciale nullo altrimenti
  - **Min(S)/Max(S)**: recuperare il minimo/massimo valore presente in S
  - **Predecessor(S, k)/Successor(S, k)**: recuperare l'elemento presente in S che precederebbe/seguirebbe quello di valore k se S fosse ordinato
- **Operazioni di manipolazione**
  - **Insert(S, k)**: inserire un elemento di valore k in S
  - **Delete(S, k)**: eliminare da S l'elemento di valore k

Le differenti strutture dati che vedremo, se da un lato hanno in comune la **capacità di memorizzare insiemi dinamici**, dall'altro differiscono anche profondamente fra loro per le **proprietà che le caratterizzano**.

Sono proprio le proprietà della struttura dati ad essere l'**elemento determinante nella scelta della struttura** da effettuare quando si deve progettare un algoritmo per risolvere un problema.

Un esempio lo abbiamo già incontrato: la struttura dati Heap (sezione 7.7.1) , senza la quale l'algoritmo Heapsort non potrebbe nemmeno essere pensato.

8.1 Array Ordinati e Disordinati

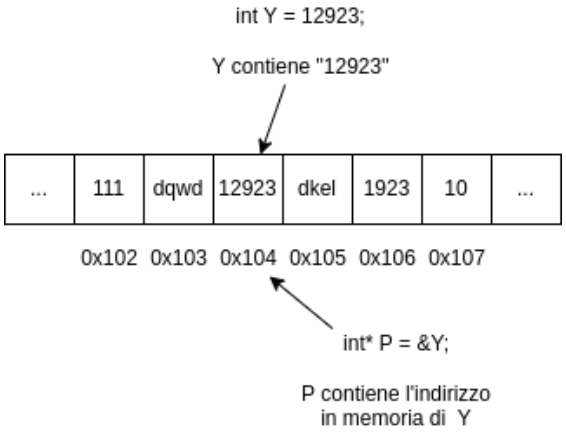
Abbiamo già visto numerose volte nei capitoli precedenti l'uso degli array come **lista statica di elementi**, sulla quale abbiamo svolto operazioni di ordinamento e di ricerca. Tuttavia, le varie operazioni di interrogazione e di manipolazione assumono un costo ben diverso a seconda dell'ordinamento dell'array stesso:

- **Array Disordinato**:
  - **Search**: è necessario scorrere l'array per trovare l'elemento, dunque un costo pari a $\Theta(n)$
  - **Min/Max**: è necessario scorrere l'array per trovare l'elemento, dunque un costo pari a $\Theta(n)$
  - **Predecessor/Successor**: è necessario scorrere l'array per trovare l'elemento, dunque un costo pari a $\Theta(n)$
  - **Insert**: inserimento nella prima posizione libera, dunque un costo pari a $\Theta(1)$
  - **Delete**: eliminazione e scambio con l'ultimo, dunque un costo pari a $\Theta(1)$
- **Array Ordinato**:
  - **Search**: essendo ordinato, possiamo usare la ricerca binaria, dunque un costo pari a $O(log(n))$
  - **Min/Max**: si prende il primo o ultimo elemento, dunque un costo pari a $\Theta(1)$
  - **Predecessor/Successor**: si prende l'elemento precedente o seguente, dunque un costo pari a $\Theta(1)$
  - **Insert**: è necessario ricercare la posizione in cui effettuare l'inserimento, per poi scorrere a destra gli elementi maggiori, dunque un costo pari a $\Theta(n)$
  - **Delete**: è necessario ricercare la posizione in cui effettuare l'inserimento, per poi scorrere a sinistra gli elementi maggiori, dunque un costo pari a $\Theta(n)$

Già da questo semplice confronto delle **due strutture dati più semplici in assoluto**, si può dedurre come **non abbia senso dire che una struttura è migliore di un'altra**, poiché le strutture dati possono essere più o meno adatte ad un certo algoritmo a seconda delle necessità dell'algoritmo stesso.

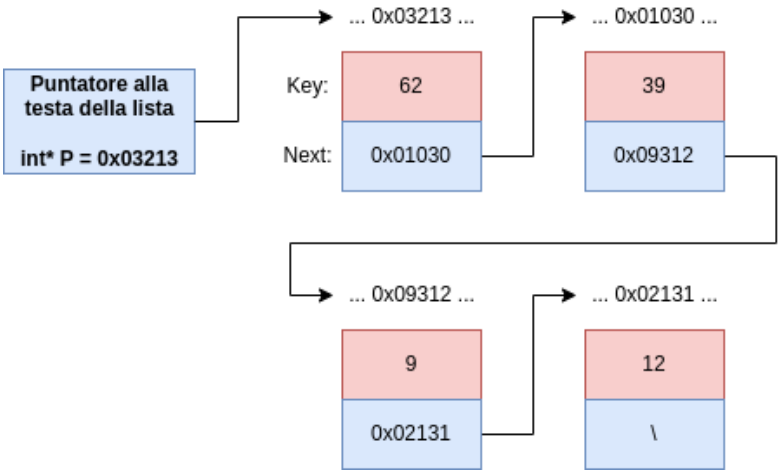
8.2 Liste puntate

Un **puntatore** è una variabile che, a differenza delle normali variabili, assume come valore un **indirizzo di memoria**, e non un valore in se e per se. Una normale variabile fa quindi riferimento ad un valore **direttamente**, mentre un puntatore lo fa **indirettamente**, puntando all'indirizzo in memoria che effettivamente contiene tale valore.



Una **lista puntata** è una struttura dati nella quale gli elementi sono organizzati in **successione** e dove ogni elemento è un **record a due campi**:

- **Un campo Key**: contenente l'**informazione vera e propria** dell'elemento. Viene rappresentato con la notazione **p -> key**.
- **Un campo Next**: contiene un **puntatore all'elemento successivo della lista**. Nel caso dell'ultimo elemento della lista, il puntatore sarà un **valore null** (rappresentato col simbolo \). Viene rappresentato con la notazione **p -> next**.



Gli **array**, ordinati e non, godono di un **accesso diretto**, dove per accedere ad un dato basta conoscerne la posizione nell'array, risultando quindi in un costo di $\Theta(1)$.

Nelle **liste puntate**, invece, la successione degli elementi viene implementata mediante un collegamento esplicito da un elemento ad un altro (ossia il puntatore), rendendo quindi possibile solo l'**accesso sequenziale**. Dunque, l'accesso a qualsiasi dato in una lista ha un costo proporzionale alla sua posizione, che nel caso peggiore è $\Theta(n)$.

Poiché l'unico accesso possibile è quello sequenziale, è intuitivo pensare che la **maggior parte delle operazioni svolte su una lista puntata abbia un costo** $O(n)$:

- **Search**: nel caso peggiore, l'elemento cercato si trova in fondo alla lista, dunque con un costo di $O(n)$

```
def Search (p: punt. alla testa; k: valore):
    p_corr = p
    while ((p_corr != NULL) and (p_corr -> key != k))
        p_corr = p_corr -> next
    return p_corr
```

- **Insert in testa**: poiché si tratta di un inserimento in una posizione qualsiasi, l'elemento viene aggiunto direttamente in testa alla lista, dunque con un costo di $\Theta(1)$

```
def Insert_in_testa (p: punt. alla testa; k: punt. a elem. da ins.):
    if (k != None)
        k->next = p
    p = k
    return p
```

- **Insert dopo un elemento**: inserimento nella posizione successiva ad da un puntatore già presente nella lista. Nel caso peggiore, il puntatore dopo cui inserire l'elemento si trova in ultima posizione, dunque con un costo di $O(n)$

```
def Insert_Dopo_d(p: punt. alla testa; k: punt. a elem. da ins.;
                  d: punt. dopo cui ins.):

    if d != None:
        k->next = d->next;
        d->next = k;
        return p
    else:
        return None
```

- **Delete**: per cancellare un elemento, è prima necessario trovarlo nella lista, dunque il costo sarà lo stesso del Search, ossia $O(n)$.

```
def Delete (p: punt. alla testa; k: punt. a elem. da canc.):
    if (k != None):         #se k=null non c'è niente da cancellare
        if k = p:           #cancella 1° elem
            p = p->next
            return p
        p_corr = p
        while (p_corr-> next 1= k)
           p_corr = p_corr-> next      #p_corr punta a elem. che prec. k
        p_corr-> next = k-> next
    return p
```

Siccome le **liste puntate sono strutture dati inerentemente ricorsive**, tutti gli algoritmi proposti possono essere anche in **versione ricorsiva**.

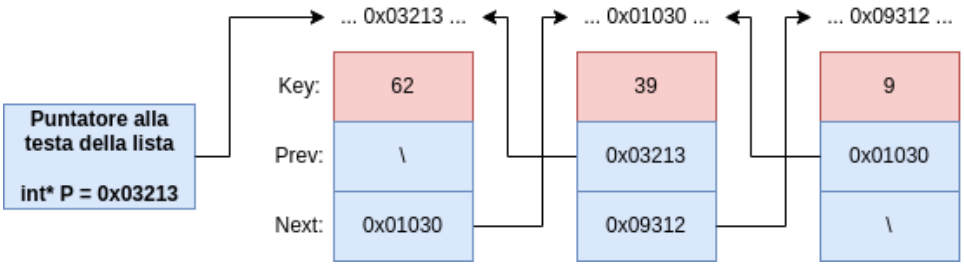
Ad esempio, la cancellazione può essere implementata come:

```
def Delete_Ric(p: punt. alla testa; k: punt. a elem. da canc.)
    if p==k:
        p = p->next
    else:
        p->next = Delete_Ric(p->next, k)
    return p
```

Liste doppiamente puntate

Alcuni problemi riscontrati nelle **liste puntate semplici** (ad esempio complessità lineare nella cancellazione) possono essere risolti organizzando la struttura dati in modo che da ogni suo elemento si possa accedere **sia all'elemento che lo segue che a quello che lo precede nella lista**, quando essi esistono.

Tale struttura dati si chiama **lista doppia o lista doppiamente puntata**, dove il puntatore all'elemento precedente viene rappresentato con la notazione **p -> prev**.



8.3 Pile e Code

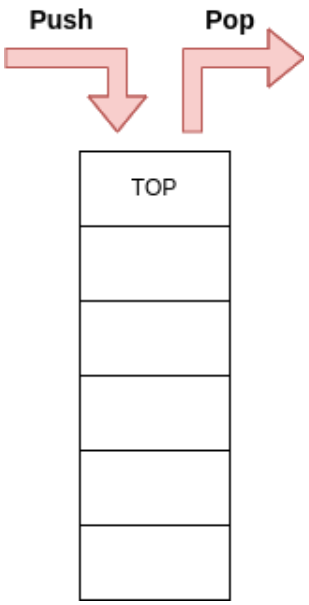
Le Pile

Una **pila** è una struttura dati basata sul comportamento **LIFO (The last in is the first out, ossia l'ultimo entrato è il primo ad uscire)**.

In altre parole, in una pila gli elementi vengono prelevati nell'**ordine inverso rispetto a quello col quale vi sono stati inseriti**, proprio come accadrebbe in una pila di piatti o di sedie: per prendere un elemento in fondo alla pila dobbiamo prima rimuovere tutti quelli al di sopra di esso.

Un esempio di utilizzo di questa struttura dati è la **pila di sistema** (ossia lo **stack di memoria**), con la quale vengono gestite le chiamate a funzione (ricorsive e non).

Per sua natura stessa, su una pila possono essere eseguite **solo due operazioni**, entrambe svolte sul **top** della pila, ossia la sua cima:



- **Push**, ossia l'inserimento in cima di un nuovo elemento, per tanto il suo costo sarà $\Theta(1)$.

```
def Push(top: punt.; e: punt. a e. da ins.):
    e->next = top
    top = e
    return top
```

- **Pop**, ossia la rimozione dalla cima dell'ultimo elemento inserito, per tanto il suo costo sarà $\Theta(1)$.

```
def Pop (top: punt):
    if (top==None):
        return None
    e = top
    top = e->next
    e->next=None
    return e, top
```

Le Code

Diversamente dalla pila, una **coda** una struttura dati basata sul comportamento **FIFO (The first in is the first out, ossia il primo entrato è il primo ad uscire)**.

In altre parole, in una coda gli elementi vengono prelevati nell'**ordine d'inserimento**, proprio come accadrebbe in una coda di persone in attesa ad uno sportello.

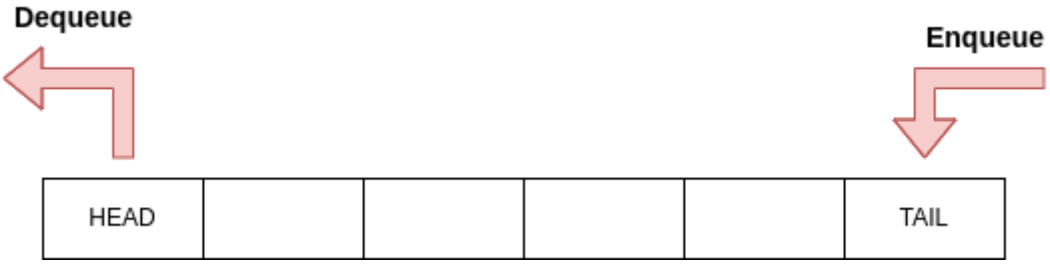
Come per le pile, anche sulle code possono essere eseguite solo due operazioni:

- **Equeue**, ossia l'inserimento in coda di un nuovo elemento (**tail**)

```
def Enqueue (head: punt.; tail: punt.; e: punt. a elem. da ins.):
    if (tail == NULL):  #la coda è vuota
        tail = e
        head = e
    else:
        tail->next = e
        tail = e
    return head, tail
```

- **Dequeue**, ossia l'estrazione dell'elemento in testa (**head**)

```
def Dequeue (head: punt.; tail: punt.):
    if (head == None):  #la coda è vuota
       return none
    e = head
    head = e->next
    if (head == None):
        tail = None
    return head, tail, e
```



Coda con Priorità

La **coda con priorità** è una variante della coda. Come nella coda, l'inserimento avviene ad un'estremità e l'estrazione avviene all'estremità opposta.

A differenza della coda, la **posizione di ciascun elemento** non dipende dal momento in cui è stato inserito, ma **dipende dal valore di una determinata grandezza, detta priorità**, la quale in generale è associata ad uno dei campi presenti nell'elemento stesso.

Quindi, gli elementi di una coda con priorità sono collocati in **ordine crescente** (o decrescente, a seconda dei casi) **rispetto alla grandezza considerata come priorità**.

8.4 Alberi

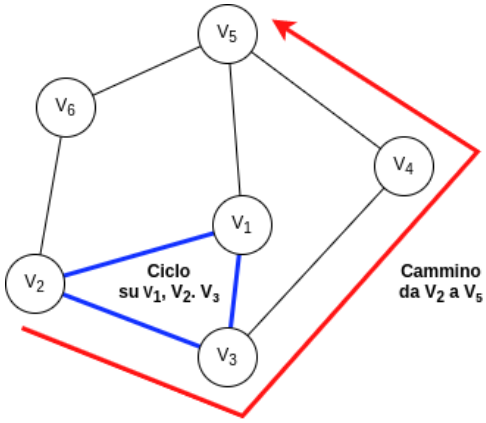
Un **albero** è una struttura dati estremamente **versatile**, utile per modellare una grande quantità di situazioni reali e progettare le relative soluzioni algoritmiche.

Per dare la definizione formale di albero è necessario prima fornire alcune definizioni relative ad un'altra struttura dati, il **grafo**:

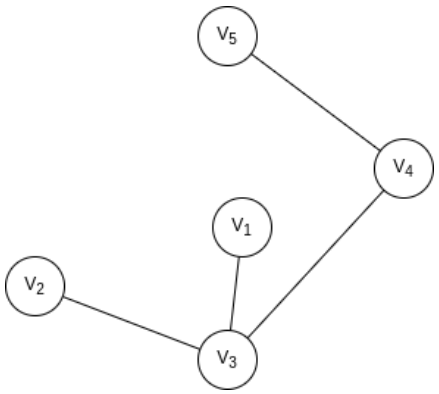
Un **grafo** $G = (V, E)$ è costituito da una **coppia di insiemi**:

- Un insieme finito V di **nodi** o **vertici**
- Un insieme finito $E \subseteq V \times V$ di coppie non ordinate di nodi, dette **archi** o **spigoli**
- Un **cammino** è una sequenza $(v_1, v_2, ..., v_k)$ di nodi distinti di V tale che (v_i, v_{i+1}) sia un arco di E per ogni $1 \leq i \leq k-1$
- Se nel cammino $(v_1, v_2, ..., v_k)$ i nodi v_k e v_1 coincidono, si parla di **ciclo**.
- Si dice che un grafo è **connesso** se, per ogni coppia di nodi (u, v), esiste un cammino tra u e v.

Una volta definita la struttura dati grafo, possiamo definire il suo diretto discendente, ossia la **struttura albero**, corrispondente ad un **grafo aciclico e connesso**.



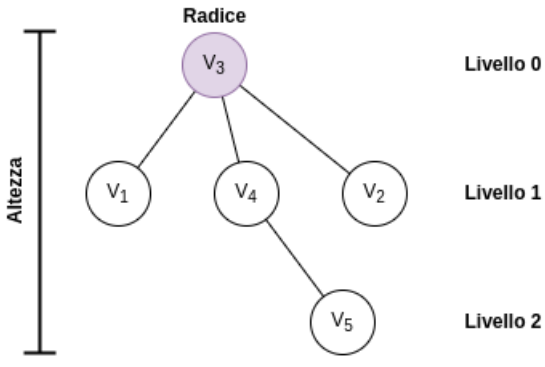
Grafo **ciclico** e **connesso**



Grafo **aciclico** e **connesso** (**Albero**)

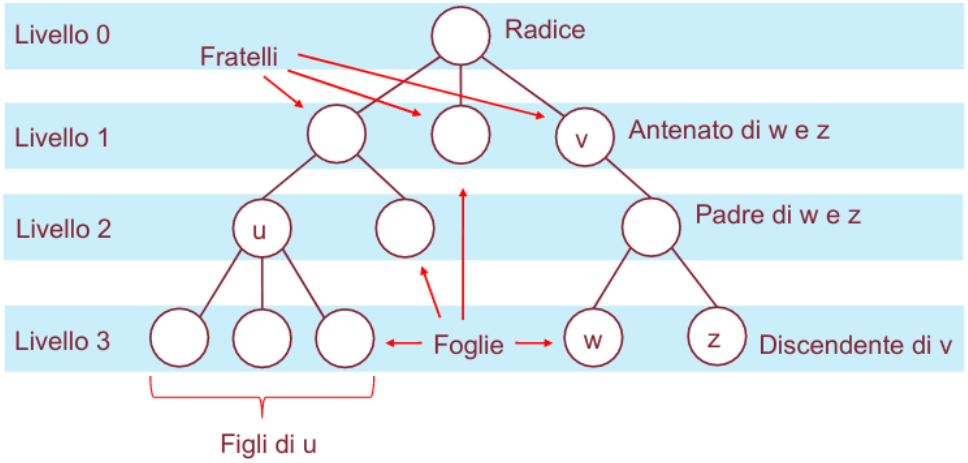
Alberi radicati

Un **albero radicato** è un albero in cui si distingue un nodo particolare tra gli altri, detto **radice**. L'albero radicato si può rappresentare in modo tale che i cammini da ogni nodo alla radice seguano un percorso dal **basso verso l'alto**, come se l'albero venisse, in qualche modo, "appeso" per la radice. I **nodi** sono organizzati in **livelli** e l'**altezza** di un albero radicato è la lunghezza del cammino più lungo dalla radice ad una foglia.



In un albero radicato distinguiamo i vari **nodi** con varie terminologie:

- Dato un qualunque nodo V di un albero radicato che non sia la radice, il **primo nodo che si incontra sul cammino da V alla radice** viene detto **padre di V**
- I nodi che hanno lo **stesso padre** sono detti **fratelli**, mentre tutti i nodi che ammettono V come padre sono detti **figli di V**. Se un nodo non ha figli, esso viene detto **foglia**
- Ogni nodo sul cammino da V alla radice viene detto **antenato di V**, mentre tutti i nodi che ammettono V come antenato vengono detti **discendenti di V**.



Inoltre, un albero radicato viene detto **ordinato**, se viene attribuito **un qualche ordine ai figli di ciascun nodo**: se un nodo ha k figli, allora vi è un figlio che viene considerato primo, uno che viene considerato secondo, ..., uno che viene considerato k-esimo.

8.4.1 Alberi binari

Una particolare **sottoclasse di alberi radicati e ordinati** è quella degli **alberi binari**, dove o**gni nodo ha massimo due figli**. Poiché sono alberi ordinati, i due figli di ciascun nodo si distinguono in **figlio sinistro** e **figlio destro**.

Un albero binario nel quale **tutti i livelli, tranne l'ultimo, contengono due figli** è chiamato **albero binario completo**, dove:

- L'**altezza** dell'albero è h
- Il **numero dei nodi di ogni livello** è 2^i, dove i è il livello
- Il **numero di foglie** è 2^h
- Il **numero de nodi interno** (ossia i nodi non foglie) è

$$\sum_{k=0}^{h-1} 2^k = \frac{2^h - 1}{2 - 1} = 2^h - 1$$

- Il **numero totale di nodi** dell'albero è

$$2^h + 2^h - 1 = 2^{h+1} - 1$$

Considerando n come il **numero totale di nodi** dell'albero, otterremo che l'**altezza** h dell'albero sarà:

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

$$log(n+1) = h + 1$$

$$log(n+1) - 1 = h$$

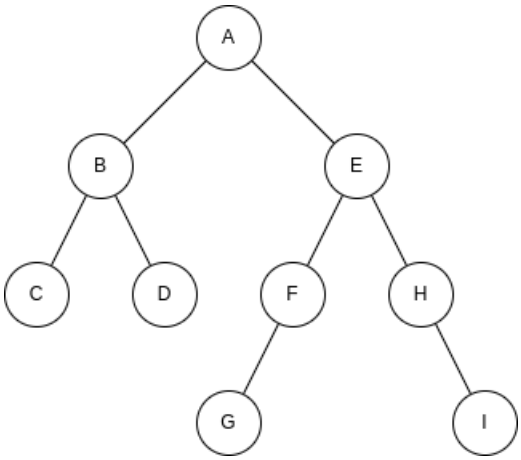
$$log(n+1) - log(2) = h$$

$$h = log\left(\frac{n+1}{2}\right)$$

Quindi, possiamo dire che l'**altezza** sia $h = O(log(n))$.

Rappresentazione e memorizzazione degli alberi

Vediamo adesso i **tre modi** con cui è possibile **rappresentare** e **memorizzare** il seguente albero binario:

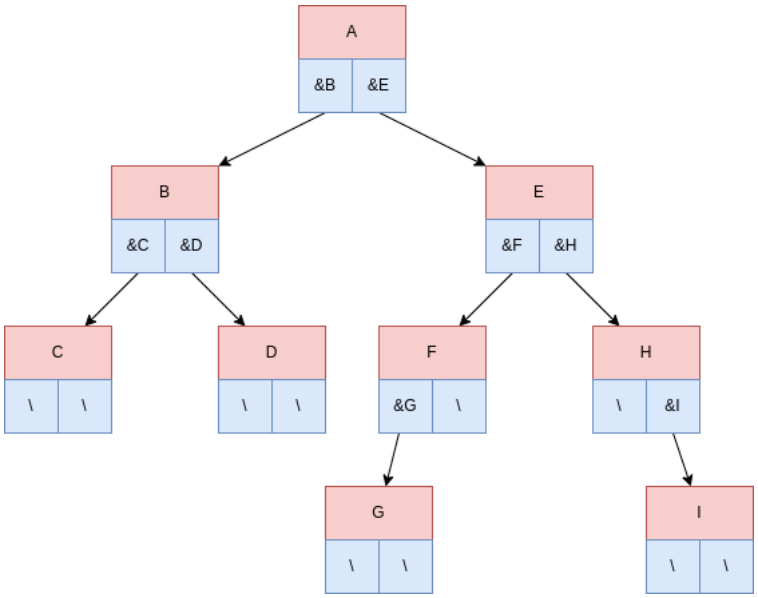


- **Tramite puntatori**:

  Il modo più naturale di **rappresentare** e **gestire** gli alberi binari è per mezzo dei **puntatori**, dove ogni singolo nodo è costituito da un **record** contenente:

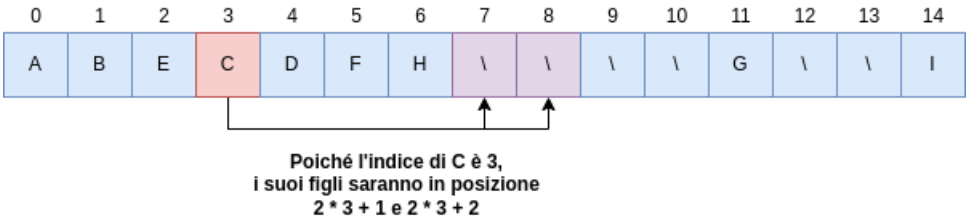
  - **Key**: le opportune informazioni pertinenti al nodo stesso
  - **Left**: il puntatore al figlio sinistro (oppure **null** se il nodo non ha figlio sinistro)
  - **Right**: il puntatore al figlio destro (oppure **null** se il nodo non ha figlio destro)

  Dunque, similmente alla testa delle liste, si accede all'albero per mezzo del **puntatore alla radice**.



- **Tramite indici posizionali**: Come nella struttura dati **heap**, i nodi vengono memorizzati in un **array**, nel quale la **radice** occupa la posizione di indice 0 ed i **figli sinistro e destro del nodo in posizione** i si trovano rispettivamente nelle **posizioni** $2i + 1$ **e** $2i + 2$.

  Tale rappresentazione, tuttavia richiede di conoscere in anticipo l'**altezza massima** dell'albero, poiché un array è una struttura dati **statica**.

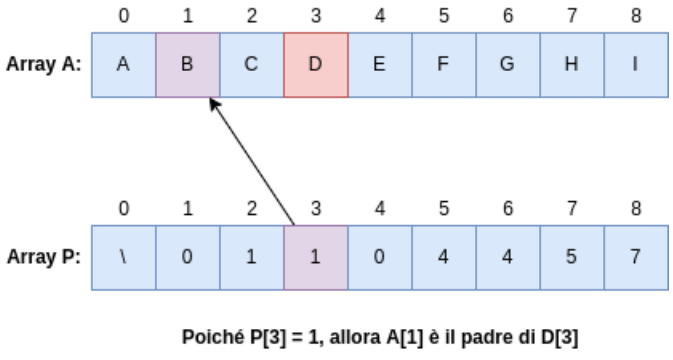


- **Tramite vettore dei padri**:

  Vengono utilizzati **due array**: **uno contenente i nodi dell'albero** A ed **uno contenente il padre di ciascun nodo** P**.**

  In questo modo, l'elemento $A[i]$ conterrà il **valore di un nodo**, mentre l'elemento $P[i]$ conterrà **l'indice del padre del nodo** i **nell'albero**.

  Questo metodo di memorizzazione funziona senza alcuna modifica anche per **alberi n-ari**, in cui cioè ogni nodo può avere un **numero qualunque di figli**.



8.4.2 Visite di un albero

Tipico degli alberi è l'**accesso progressivo a tutti i nodi**, uno dopo l'altro, al fine di poter effettuare una specifica operazione su ciascuno di essi.

A differenza di array e liste, tale operazione non può essere effettuata con una semplice iterazione per via della struttura più articolata degli alberi. Questa operazione viene chiamata **visita dell'albero**.

Facendo riferimento all'**ordine** col quale si accede ai nodi dell'albero, è evidente che esiste **più di una possibilità** a seconda del **momento in cui si svolge l'operazione richiesta** sul nodo stesso.

Nell'ambito degli **alberi binari**, possiamo identificare **tre tipi di visite**:

- **Visita in pre-order**: si accede al nodo prima di proseguire la visita nei suoi sottoalberi

```
def visita_pre_order(p):
    if (p != None)
        fun(p) #accesso al nodo e operazioni conseguenti
        visita_pre_order(p->left)
        visita_pre_order(p->right)
    return
```

- **Visita in-order**: il nodo è visitato dopo la visita del sottoalbero sinistro e prima di quella del sottoalbero destro

```
def visita_in_order(p):
    if (p != None)
        visita_in_order(p->left)
        fun(p) #accesso al nodo e operazioni conseguenti
        visita_in_order(p->right)
    return
```

- **Visita post-order**: il nodo è visitato dopo entrambe le visite dei sottoalberi.

```
def visita_post_order(p):
    if (p != None)
        visita_post_order(p->left)
        visita_post_order(p->right)
        fun(p) #accesso al nodo e operazioni conseguenti
    return
```

Il **costo di tutte e tre le visite è lo stesso** ed riconducibile alla seguente equazione, dove k è il numero di nodi dell'sottoalbero sinistro, mentre $n-k-1$ è il numero nodi di quello destro:

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

Analizziamo quindi il suo **caso migliore** e il suo **caso peggiore**:

- **Caso migliore:** si tratta di un **albero completo**, dunque ogni nodo (tranne le foglie) possiede **esattamente due figli**. In questo caso, ogni sottoalbero sinistro e ogni sottoalbero destro avranno $\frac{n-1}{2}$ nodi, dunque:

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

  Poiché ricadiamo nel **caso 1 del metodo principale**, il costo del caso migliore sarà $T(n) = \Theta(n)$

- **Caso peggiore:** si tratta di un **albero estremamente sbilanciato**, dove tutti i nodi discendenti della radice sono agglomerati nel sottoalbero sinistro, dunque quando $n - k - 1 = 0$, o nel sottoalbero destro, dunque quando $k = 0$.
  - Se $k = 0$ si ha $n - k - 1 = n - 1$:

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

  - Se $n - k - 1 = 0$ si ha $k = n - 1$:

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

  In entrambi i casi, possiamo utilizzare il metodo iterativo per ricondurre facilmente il costo a $\Theta(n)$.

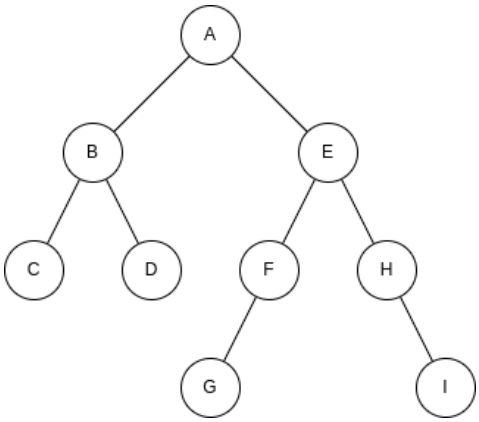
- **Caso generale:** poiché si ha che l'equazione è sia $O(n)$ (per via del **caso peggiore**), sia $\Omega(n)$ (per via del **caso migliore**), concludiamo che il suo costo sia sempre $\Theta(n)$. Difatti, poiché una visita consiste nell'**analizzare tutti i nodi di un albero** e poiché n corrisponde al **numero di nodi totali**, viene logico che il **costo di una visita sia sempre** $\Theta(n)$.

E se volessimo effettuare una **visita per livelli**, dalla radice in giù? Nessuna delle visite ricorsive che abbiamo illustrato per gli alberi implementati mediante puntatori permette di farlo. È necessario utilizzare una **coda d'appoggio**, nella quale inserire opportunamente i nodi, estraendoli poi per visitarli.

```
def visita_per_livelli (r; head, tail):
    if (r == None): return
    Enqueue(head, tail, r)
    while (!CodaVuota(head)):
        p = Dequeue(head, tail)
        fun(p) #accesso al nodo e operazioni conseguenti
        if (p->left != None): Enqueue(head, tail, p->left)
        if (p->right != None): Enqueue(head, tail, p->right)
    return
```

Esempi di visita:

1. Consideriamo il seguente albero già visto precedentemente:



   Immaginiamo di voler stampare tutti i nodi di questo albero. A seconda della **tipologia di visita** applicata otteniamo un **output diverso**:

   - **Visita pre-order**:
     A B C D E F G H I
   - **Visita in-order**:
     C B D A G F E H I
   - **Visita in-order**:
     C D B G F I H E A

2. Vogliamo contare il numero di nodi di un albero binario.

   Poiché il numero di nodi di un sottoalbero corrisponde al **numero di nodi dei due sotto-sottoalberi sinistro e destro sommati alla radice stessa del sottoalbero**, ne deduciamo che sia prima necessario chiamare ricorsivamente la funzione su entrambi i due sottoalberi, per poi effettuare la somma.

   Il risultato, quindi, è l'applicazione di una **visita post-order**:

```
def Calcola_n (p):
    if (p != None):
        num_l = Calcola_n (p->left)
        num_r = Calcola_n (p->right)
        num = num_l + num_r + 1     #accesso al nodo
        return num
    return 0
```

   Che possiamo anche comprimere nel seguente codice:

```
def Calcola_n (p):
    if (p != None):
        return Calcola_n (p->left) + Calcola_n (p->right) + 1
    return 0
```

3. Vogliamo ricercare un nodo in un albero binario.

   Immaginando di star effettuando noi la ricerca, ci viene logico pensare che, **prima** di controllare i nodi successivi, venga controllato il **nodo attuale**.

   Il risultato, quindi, è l'applicazione di una **visita pre-order**:

```
def Cerca (p):
    if (p != None):
        if p->info == k return TRUE
    else:
        if Cerca(p->left,k) == TRUE:
            return TRUE
    else:
        return Cerca(p->right,k)
    return FALSE
```

4. Vogliamo calcolare l'altezza di un albero binario.

   Poiché l'altezza di un sottoalbero corrisponde all'**altezza dei suoi sotto-sottoalberi incrementata di uno**, ne deduciamo che sia necessario chiamare ricorsivamente la funzione prima di effettuare l'incremento.

   Il risultato, quindi, è l'applicazione di una **visita post-order**:

```
def Calcola_h (p):
    if (p==None):
        return -1   #albero vuoto
    if (p->left==None) and (p->right==None):
        return 0
    h = max(Calcola_h(p->left), Calcola_h(p->right))
    return h + 1
```

8.4.3 Alberi binari di ricerca

Un **albero binario di ricerca (ABR)** è una versione specifica dell'albero binario nel quale vengono mantenute le seguenti **proprietà aggiuntive**:

- Il valore della chiave contenuta in ogni nodo è **maggiore** della chiave contenuta in **ciascun nodo del suo sottoalbero sinistro (se esiste)**. Dunque, il **valore minimo assoluto** dell'albero è il nodo **più a sinistra**.
- Il valore della chiave contenuta in ogni nodo è **minore** della chiave contenuta in **ciascun nodo del suo sottoalbero destro (se esiste)**. Dunque, il **valore massimo assoluto** dell'albero è il nodo **più a destra**.

Non è detto che il minimo e il massimo siano necessariamente una foglia dell'albero

Ordinamento dei nodi

Immaginiamo di voler stampare i nodi di un albero binario di ricerca (ABR) in **ordine crescente**.

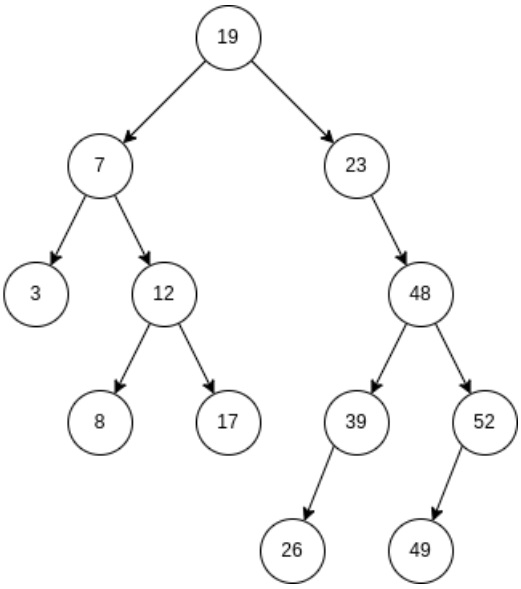
Ricordando che, secondo le **proprietà degli alberi binari di ricerca**, i valori minori di un nodo sono nel suo sottoalbero sinistro e mentre i valori maggiori sono nel sottoalbero destro, possiamo applicare una semplice **visita in-order** per stampare l'intero albero in ordine crescente:

```
def Stampa_crescente(p):
    if (p != None)
        Stampa_crescente(p->left)
        print(p->val)
        Stampa_crescente(p->right)
```

Potenzialmente, quindi, potremmo considerare l'uso di un ABR come un **algoritmo di ordinamento**, costituito dalla costruzione dell'ABR (il cui costo ancora non conosciamo) e dalla sua visita in-order (il cui costo sappiamo essere $\Theta(n)$).

Visita in-order:

3, 7, 8, 12, 17, 19, 23, 26, 39, 39, 48, 49, 52



Ricerca di un nodo

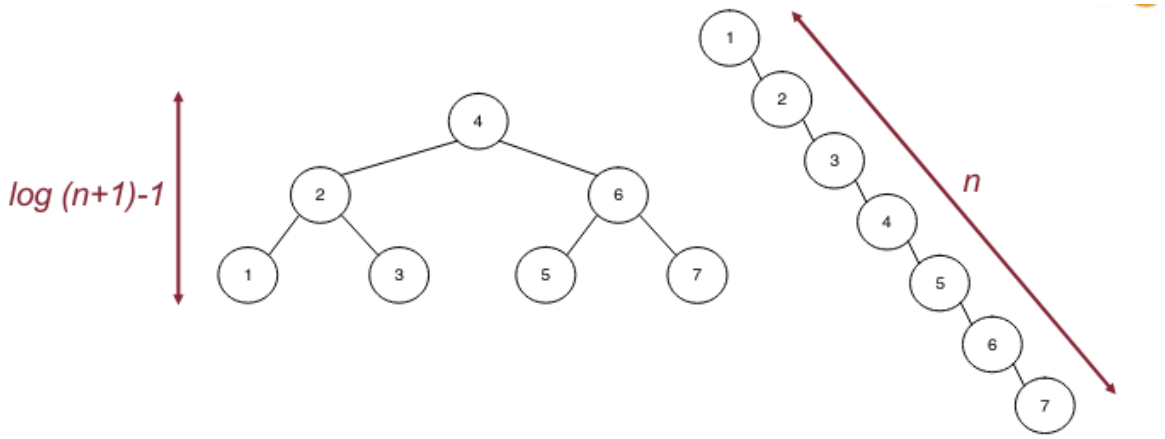
Per via delle sue proprietà, inoltre, notiamo come una **ricerca su un ABR** sia concettualmente simile alla **ricerca binaria**: si esegue una discesa dalla radice che viene guidata da un **confronto (maggiore o minore) tra i valori memorizzati nei nodi** che si incontrano lungo il cammino:

```
def ABR_search(p,k):
    if (p == None) or (p->key = k):
        return p
    if (k < p->key):
        return ABR_searchRic(p->left, k)
    else:
        return ABR_searchRic(p->right, k)
```

Tuttavia, a differenza della ricerca binaria, in questo caso **non riusciamo a garantire un costo computazionale pari a** $O(log(n))$, poiché il costo dell'algoritmo di ricerca è $O(h)$, dove h è l'**altezza dell'ABR**:

- Nel **caso peggiore**, ossia un albero completamente sbilanciato (dove tutti i nodi sono agglomerati a sinistra o a desra), corrisponde a n, dunque il costo $O(n)$
- Nel **caso migliore**, ossia un albero completamente sbilanciato (dove tutti i nodi sono agglomerati a sinistra o a desra), corrisponde a $log(n+1)-1$, dunque il costo $\Omega(log(n))$

Quindi, per **garantire** che il costo computazionale della ricerca su un ABR sia $O(log(n))$, è necessario mettere in campo qualche tecnica che ci permetta di tenere sotto controllo la crescita dell'altezza, ossia il **bilanciamento in altezza**.



Inserimento di un nodo

Come nella ricerca, si esegue una discesa che viene guidata dai **confronti sulle chiavi** memorizzate nei nodi che si incontrano lungo il cammino: quando si arriva al punto di voler proseguire la discesa verso un **puntatore vuoto** (None), si interrompe la discesa e si **aggiunge il nuovo nodo** contenente la chiave da inserire in quella posizione.

```
def ABR_insert (p,k):
    y,x = None,p        #y punta sempre al padre di x
    z = NodoABR(k)      #nuovo nodo con k come chiave

    while (x != None)   #discesa alla prima pos. disponib.
        y = x
        if z->key < x->key:
            x = x->left
        else:
            x = x->right

    if (y==None):       #se l'albero era inizialmente vuoto
        p = z
    else:
        if (z->key < y->key):
            y->left = z
        else:
            y->right = z

    z->parent = y       #collegam. padre - nodo da inser.
    return p            #p potrebbe essere cambiato
```

Eliminazione di un nodo

Nell'eliminazione di un nodo da un ABR, si possono verificare **tre casi**:

- **Caso 1**: se il nodo è una **foglia**, lo si elimina ponendo a None l'opportuno campo nel suo nodo padre
- **Caso 2**: se il nodo ha **un solo figlio**, lo si elimina collegando direttamente suo padre e il suo unico figlio, sovrascrivendo il collegamento tra suo padre e il nodo eliminato stesso
- **Caso 3**: se il nodo ha **entrambi i figli**, è necessario **riaggiustare l'albero** significa trovare un nodo da collocare al posto del nodo che va eliminato, così da **mantenere l'albero connesso** e da **garantire il mantenimento delle proprietà fondamentali** degli ABR.

  Per ottenere ciò, lo si sostituisce col **predecessore** (o col **successore**), che va quindi eliminato dalla sua posizione originale (operazione che questa volta ricade in uno dei due casi precedenti).

8.4.4 Alberi rosso-neri

Come abbiamo già visto, per ottenere il costo computazionale minore nello svolgere operazioni su un ABR è necessario che l'albero sia il **più bilanciato possibile**, in modo che la sua altezza sia $h = O(log(n))$.

Le **tecniche di bilanciamento** sono tutte basate sull'idea di **riorganizzare la struttura dell'albero** se essa, a seguito di un'operazione di inserimento o di eliminazione di un nodo, viola determinati requisiti.

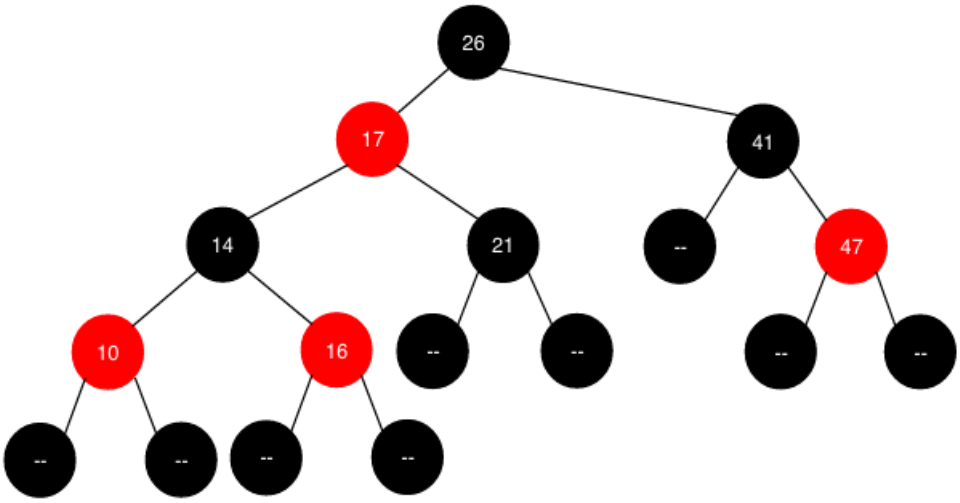
In particolare, il requisito da controllare è che, per ciascun nodo dell'albero, l'altezza dei suoi due sottoalberi non sia **"troppo differente"**. Ciò che rende non banali queste tecniche è che si vuole aggiungere agli ABR una proprietà **senza peggiorarne il costo computazionale delle operazioni**.

Un **albero rosso-nero (A-RB)** è un ABR i cui nodi hanno un campo aggiuntivo, ossia il **colore**, che può essere **solo rosso o nero**.

All'albero vengono aggiunte **foglie fittizie** (che non contengono chiavi) in modo che tutti i nodi "veri" dell'albero abbiano **esattamente due figli**.

Un A-RB, quindi, è un ABR che soddisfa le seguenti **proprietà aggiuntive**:

- Ciascun nodo è **rosso** o **nero**
- Ciascuna **foglia fittizia è nera**
- Se un nodo è **rosso** i suoi figli sono **entrambi neri**
- **Ogni cammino** da un **nodo** a ciascuna delle **foglie** del suo sottoalbero contiene lo **stesso numero di nodi neri**
- La **radice** è sempre **nera**
- Nessun cammino dalla radice ad una foglia può essere lungo più del doppio di un cammino dalla radice ad una qualunque altra foglia



Definiamo quindi come **black-height** di un **nodo x** il numero di nodi neri sui cammini dal nodo x (non incluso) alle sue foglie discendenti. La BH di un A-RB è la **BH della sua radice**, mentre il sottoalbero radicato in un qualsiasi nodo x contiene **almeno** $2 \cdot BH(x) - 1$ **nodi interni**.

Tramite le proprietà degli A-RB, possiamo garantire il seguente teorema:

Theorem 5

Un A-RB con n nodi interni ha **sempre un'altezza** h

$$h \leq 2log(n+1)$$

Dimostrazione: Sia h l'altezza dell'A-RB ed r la sua radice. Sappiamo già che $h \leq 2BH(r)$, ossia $BH(r) \geq \frac{h}{2}$. Sappiamo inoltre che il numero di nodi interni dell'A-RB, pari al numero di nodi interni nel sottoalbero radicato in r è:

$$n \geq 2^{BH(r)} - 1 \geq 2^{\frac{h}{2}} - 1$$

da cui

$$n + 1 \geq 2^{\frac{h}{2}} \quad \Longrightarrow \quad 2log(n+1) \geq h$$

Tale teorema, quindi, **garantisce** che le operazioni di:

- Ricerca di una chiave
- Ricerca del massimo o del minimo
- Ricerca del predecessore o del successore
- Inserimento di un nodo
- Eliminazione di un nodo

siano tutte eseguite con un **costo computazionale** $O(log(n))$.

Rotazioni di un A-RB

L'esigenza di **mantenere le proprietà di A-RB** implica che, dopo un inserimento o una eliminazione, la struttura dell'albero debba essere **riaggiustata**, in termini di:

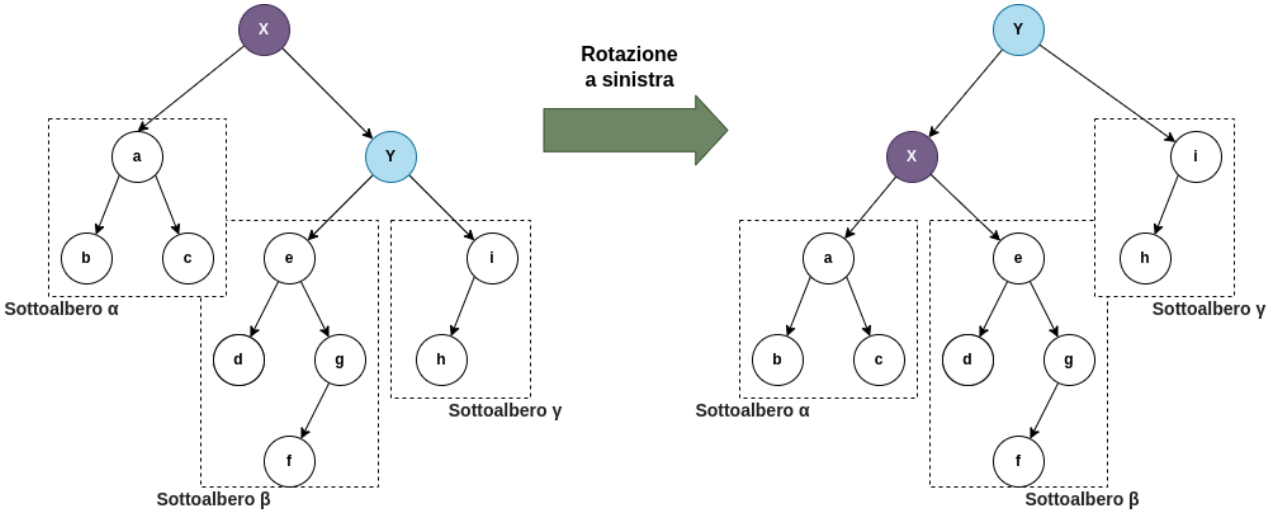
- Colori assegnati ai nodi
- Collocazione dei nodi nell'albero

A tal fine sono definite apposite operazioni, dette **rotazioni**, che permettono di ripristinare in tempo $O(logn)$ le proprietà del A-RB dopo un inserimento o una eliminazione.

Le **rotazioni** possono essere **destre** o **sinistre** e sono operazioni **locali** che non modificano l'ordinamento delle chiavi secondo la visita in-order.

Ogni rotazione viene effettuata su un **nodo x**, il quale funge da "**perno**" su cui viene effettuata l'intera rotazione: in una **rotazione a sinistra**, il **sottoalbero sinistro** β **del figlio destro del nodo x** (che chiameremo nodo y), diventa il **nuovo sottoalbero destro del nodo x stesso**, mentre il nodo x diventa il **figlio sinistro del nodo y**.

La **rotazione a destra** risulta specchiata a quella a sinistra, invertendo nodi e sottoalberi sinistri con quelli destri.




```
def A_RB_rotaz_sinistra (p, x): #x = il nodo su cui si ruota
    y = x->right
    x->right = y->left

    if x->right != None:
        x->right->parent = x
    y->left = x
    y->parent = x->parent

    if x->parent == None:
        p = y
    else:
        if x == x->parent->left:
            x->parent->left = y
        else:
            x->parent->right = y
    x->parent = y

    return p
```

Il **costo computazionale** di una rotazione risulta essere $\Theta(1)$, poiché vengono effettuati solo scambi costanti tra i nodi.

Nota: una visita in-order su un A-RB effettuata prima e dopo una rotazione mantiene lo stesso ordine di accesso ai nodi.

Inserimento in un A-RB

Per via delle sue strette regole, l'inserimento di un nodo in un A-RB risulta complesso e strutturato in **più fasi e casi**:

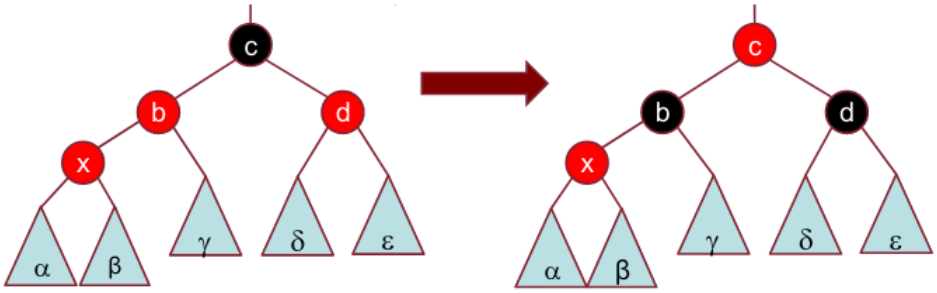
- **Fase preliminare**: ogni nodo viene inserito come in un normale albero binario di ricerca e gli viene attribuito il **colore rosso**
- **Fase di aggiustamento**: è necessario se e solo se il nodo inserito risulta essere il **figlio di un altro nodo rosso**. A questo punto, si può rientrare in una di **quattro casistiche**:

0 **Caso 0 - Il nodo x inserito risulta essere radice dell'albero**:

- Poiché la radice deve essere sempre nera, è necessario semplicemente **cambiare il colore del nodo da rosso a nero**.

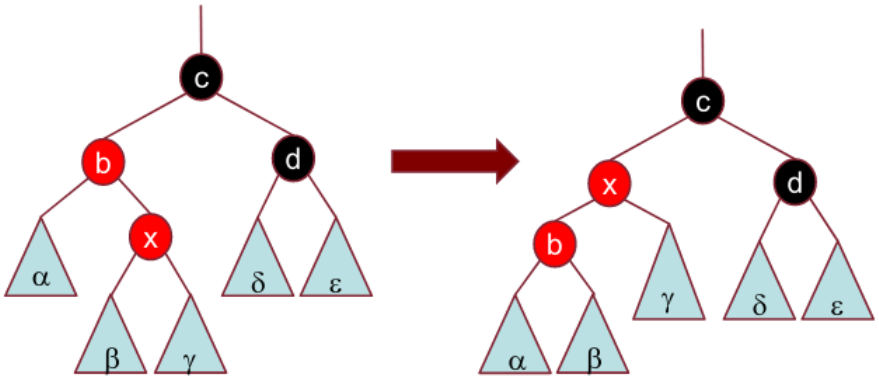
1 **Caso 1 - Il nodo x inserito risulta essere figlio di un nodo rosso e il suo zio è rosso**:

- È necessario cambiare i colori di alcuni nodi: si colorano **di nero il padre e lo zio del nodo x** e **di rosso il nonno di x**.
- La violazione risulta quindi **eliminata** oppure è stata **portata più in alto**, ricadendo sul nonno di x.



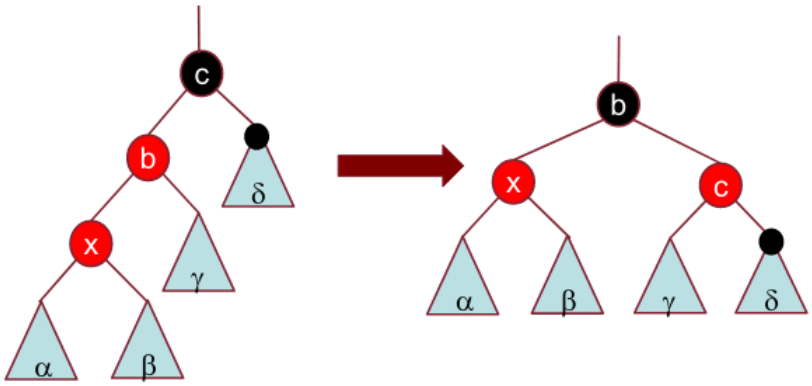
2 **Caso 2 - Il nodo x inserito risulta essere figlio destro di un nodo rosso e il suo zio è nero**:

- Si effettua una **rotazione a sinistra sul padre di x**

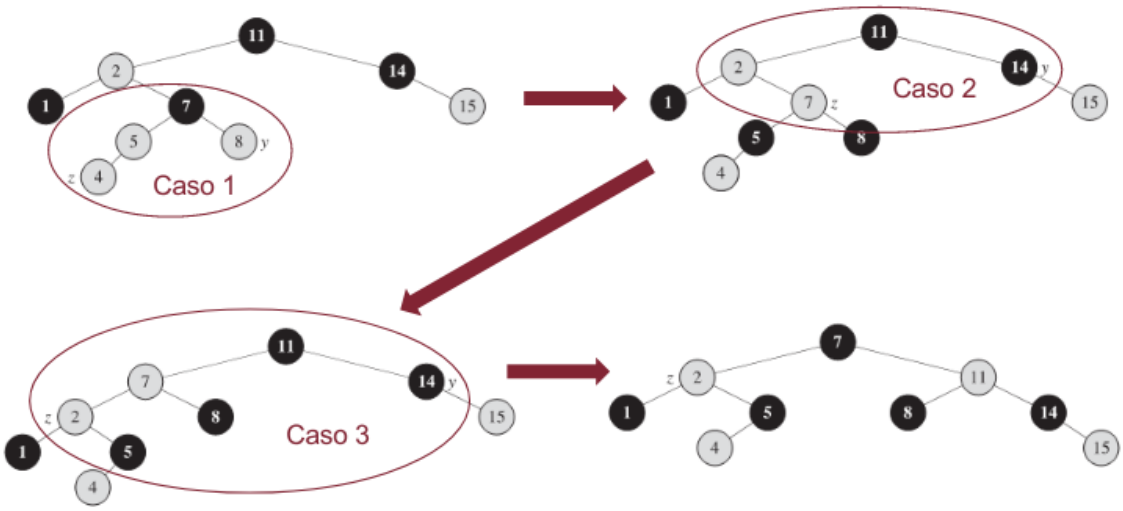


3 **Caso 3 - Il nodo x inserito risulta essere figlio sinistro di un nodo rosso e il suo zio è nero**:

- L'applicazione del Caso 2 **riconduce sempre al Caso 3**
- Si effettua una **rotazione a destra** e si **cambiano alcuni colori**, garantendo sempre la soluzione della violazione.



Esempio di inserimento completo su un A-RB



1. In ogni passaggio dell'inserimento etichettiamo come z **il nodo che viola le proprietà degli A-RB**
2. Inserendo il nodo z si ricade nel **Caso 1**, dunque cambiamo i colori necessari
3. La risoluzione del Caso 1 precedente da vita al **Caso 2**, dunque si effettua una rotazione a sinistra sul padre del nodo z
4. Poiché la risoluzione del Caso 2 da sempre vita al **Caso 3**, effettuiamo la rotazione finale che risolve la violazione

Notiamo come, dopo l'inserimento, l'A-RB risulti essere ancora **quasi completamente bilanciato**.

8.5 Dizionari

Un **dizionario** è una struttura dati che permette di gestire un insieme dinamico di dati, che di norma è un **insieme totalmente ordinato**, tramite queste tre sole operazioni:

- **Insert**: si inserisce un elemento
- **Search**: si ricerca un elemento
- **Delete**: si elimina un elemento

Quando l'esigenza è quella di realizzare un dizionario, si ricorre quindi a soluzioni specifiche:

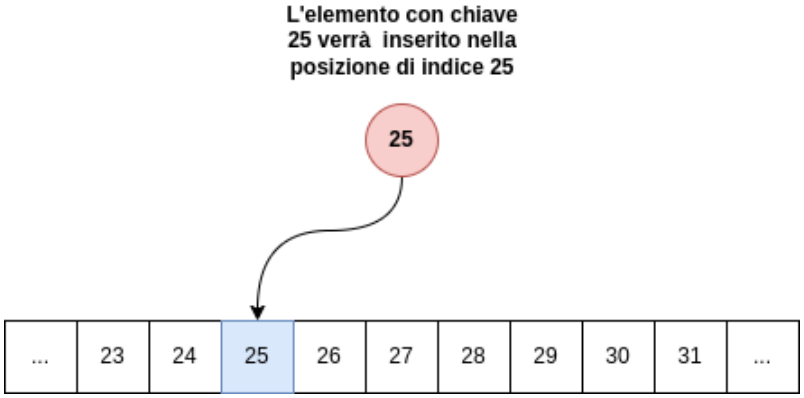
- **Tabelle ad indirizzamento diretto**
- **Tabelle hash**
- **Alberi binari di ricerca**

Prima di parlare di esse, è necessario prima introdurre delle **assunzioni** e delle **nomenclature**:

- **Insieme U**: è l'insieme universo dei **valori che le chiavi possono assumere** ed è costituito da **valori interi**
- **Valore m**: è il **numero delle posizioni a disposizione** nella struttura dati
- **Valore n**: è il **numero degli elementi da memorizzare nel dizionario** e i valori delle chiavi degli elementi da memorizzare sono tutti diversi fra loro.

8.5.1 Tabelle ad indirizzamento diretto

Un dizionario costituito da una **tabella ad indirizzamento diretto**, è un vettore nel quale **ogni indice corrisponde al valore della chiave** dell'elemento da memorizzare in tale posizione.



Affinché tale tipologia di dizionario possa funzionare, **è necessario che** $n \leq m = |U|$, dove $|U|$ indica il numero di elementi dell'insieme U, assolvendo perfettamente al compito di dizionario con **grande efficienza**.

Infatti, tutte e tre le operazioni necessarie hanno costo computazionale pari a $\Theta(1)$:

```
def Insert_Indirizz_Diretto (A; e: elem da ins.)
    A[e->key] = e
    return

def Search_Indirizz_Diretto (A; k: chiave da cerc.)
    return A[k]

def Delete_Indirizz_Diretto (A; k: chiave da canc.)
    A[k] = None
    return
```

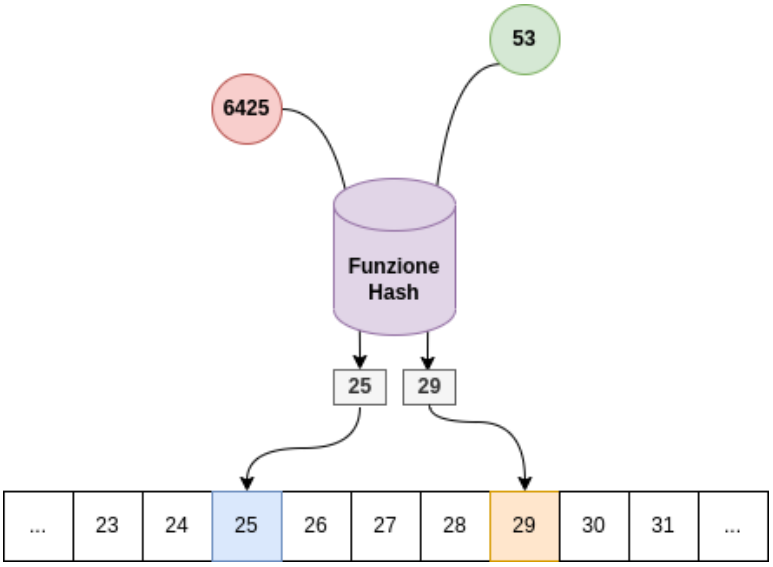
Sebbene sia estremamente efficiente quanto a costo computazionale, l'indirizzamento diretto risulta essere il meno efficiente tra tutti per quanto riguarda il **costo in memoria**:

- Poiché si ha $m = |U|$, un insieme U enorme rende impraticabile l'**allocazione in memoria** di un array lungo m
- Il numero delle chiavi effettivamente utilizzate può essere molto più piccolo di $|U|$: in tal caso vi è un **rilevante spreco di memoria**

Perciò, si ricorre spesso a differenti implementazioni dei dizionari, a meno che non ci si trovi nelle condizioni che permettono l'uso dell'indirizzamento diretto, in particolare il mantenimento di un insieme U ridotto.

8.5.2 Tabelle hash

Per risolvere il problema principale degli indirizzamenti diretti, si ricorre all'uso di una **funzione di hash**, in grado di calcolare la posizione finale di un elemento sulla base del valore della sua chiave, il quale può essere **un intero molto più grande del valore generato dalla funzione di hash**.



Tuttavia, anche se le chiavi da memorizzare sono meno di m, non si può escludere che due chiavi $k_1 \neq k_2$ siano tali per cui $hash(k_1) = hash(k_2)$, ossia che la **funzione hash restituisca lo stesso valore per entrambe le chiavi**, che quindi andrebbero memorizzate nella stessa posizione della tabella, dando vita a quello che viene definito come **collisione di hash**.

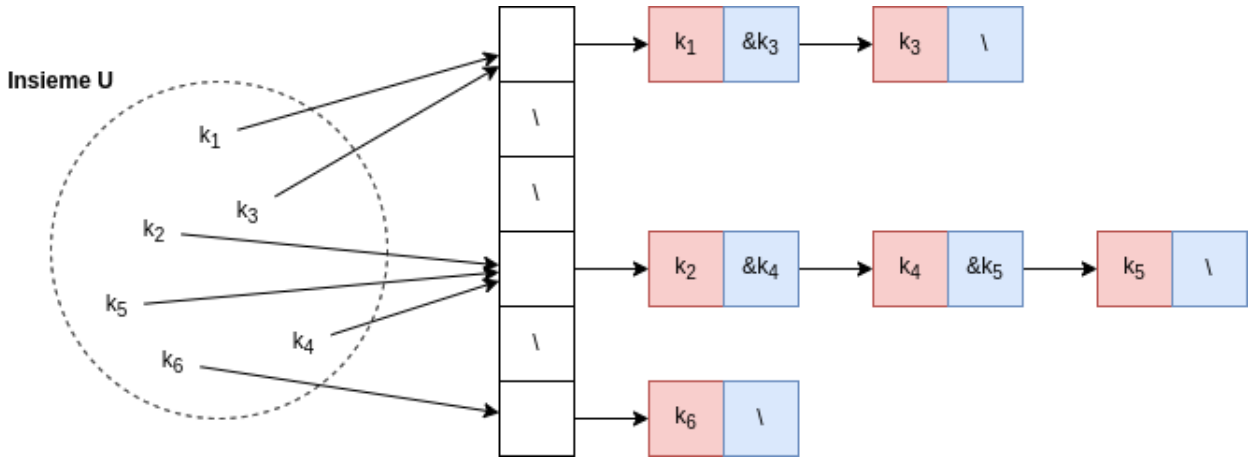
Tale fenomeno è **matematicamente inevitabile**. Dunque, possiamo solo tentare di evitarle il più possibile e risolverle con alcune strategie:

- Una buona funzione hash deve essere tale da rendere il **più equiprobabile possibile il valore risultante** dall'applicazione della funzione: tutti i valori fra 0 e $m - 1$ dovrebbero essere restituiti con uguale probabilità.
- In altre parole, la funzione dovrebbe far apparire come "casuale" il valore risultante, disgregando qualunque regolarità della chiave.
- Inoltre, la funzione deve essere **deterministica**: se applicata più volte alla stessa chiave deve fornire sempre lo stesso risultato.
- La situazione ideale è quella in cui ciascuna delle m posizioni della tabella è scelta deterministicamente con la stessa probabilità: **ipotesi di uniformità semplice della funzione hash**.

L'i**potesi di uniformità semplice** (di cui ometteremo le specifiche per poterla ottenere) minimizza il numero di collisioni, ma queste possono comunque avvenire: per quanto sia progettata bene la funzione hash, è **impossibile evitare del tutto le collisioni**, poiché, siccome si ha che $|U| > m$, è inevitabile che esistano chiavi diverse che producono una collisione.

8.5.3 Tabelle a liste di trabocco

Questa tecnica prevede di inserire **tutti gli elementi le cui chiavi mappano nella stessa posizione in una lista puntata**, detta **lista di trabocco**. Invece degli elementi in se, dunque, nell'array del dizionario vengono immagazzinati i puntatori alle teste delle liste di trabocco.



Poiché tali dizionari vengono implementati tramite l'uso di liste puntate, ne segue che il costo delle operazioni sia **strettamente legato ad esse**:

- **Inserimento**: il costo rimane $\Theta(1)$, poiché possiamo effettuare un inserimento in testa sulle liste

```
def Insert_Liste_Trabocco (A; e: elem. da ins.):
    lista_di_trabocco = A[hash(e->key)]
    insert_in_testa(lista_di_trabocco, e)
    return
```

- **Ricerca**: poiché la ricerca di un elemento in una lista puntata ha un **costo pari a** $O(i)$, dove i è la **lunghezza della lista**, ne segue che il costo di una ricerca in un dizionario di questo tipo nel **caso peggiore** risulti essere $O(n)$, dovuto all'inserimento di tutti gli elementi nella stessa identica lista di trabocco.

  Nel **caso medio**, il costo della ricerca risulta essere $O(\alpha)$, dove α è il **fattore di carico della tabella**. Considerando l'**ipotesi di uniformità semplice**, otteniamo che $\alpha = \frac{n}{m}$, dunque il costo della ricerca risulta essere $O\left(\frac{n}{m}\right)$

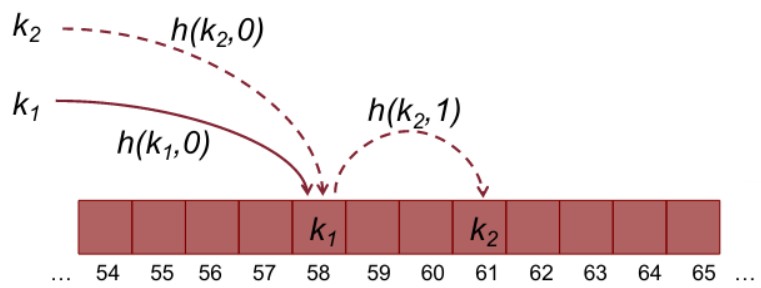
```
def Search_ListeTrabocco (A; k: chiave da cercare):
    lista_di_trabocco = A[hash(k)]
    elem = ricerca_elem(lista_di_trabocco, k)
    return elem
```

- **Eliminazione**: dipende strettamente dall'implementazione delle liste di trabocco e valgono, pertanto, **tutte le osservazioni fatte** per il costo dell'operazione di cancellazione nelle liste concatenate.

```
def Delete_ListeTrabocco (A; k: chiave da canc.)
    lista_di_trabocco = A[hash(k)]
    elimina_elem(lista_di_trabocco, k)
    return
```

8.5.4 Tabelle ad indirizzamento aperto

Al contrario delle liste di trabocco, l'idea alla base di tale tecnica, prevede l'inserimento solo all'interno dell'array stesso, calcolando la **sequenza delle posizioni da esaminare**. La **funzione hash dipende ora da 2 parametri**: la chiave k e il **numero di collisioni già trovate**. L'implementazione di tale nuova funzione hash verrà vista in seguito.



Questa tecnica è applicabile solo quando:

- $m \geq n$ (quindi il fattore di carico $\alpha = \frac{n}{m}$ non è mai maggiore di 1)
- $|U| >> m$, ossia il numero di elementi dell'insieme U è molto più grande di m

Operazioni sul dizionario

- **Inserimento**: se la posizione calcolata con $h(k, 0)$ (dove 0 è il numero di collisioni generate durante l'inserimento attuale) è **già occupata**, allora viene **ricalcolato l'hash** incrementando di 1 il numero di collisioni (dunque $h(k, 1)$), generando un valore completamente diverso dal precedente. Se anche la nuova posizione è già occupata, verrà ripetuto ancora il processo, **fino ad un massimo di** $h(k, m-1)$.

  Nel caso peggiore, il costo di tale operazione risulta essere $O(n)$ e si verifica quando tutti gli n elementi inseriti restituiscono lo stesso $h(k, 0)$. Considerando l'ipotesi di uniformità semplice, invece, il caso medio risulta essere $O\left(\frac{1}{1-\alpha}\right) = O\left(\frac{m}{m-n}\right)$
- **Ricerca**: si analizza la tabella mediante la stessa **sequenza di funzioni hash** utilizzata per l'inserimento fino a quando si incontra l'elemento cercato oppure una casella vuota, deducendo che l'elemento non è presente. Ne segue quindi che i suoi costi computazionali risultino analoghi a quelli dell'inserimento.
- **Eliminazione**: tale operazione risulta essere molto problematica, poiché nel caso in cui si vada ad eliminare l'elemento lasciando la casella vuota, ciò **spezzerebbe la catena di sequenza della funzione hash** di un qualunque elemento già inserito nella tabella.

  Una soluzione proponibile sarebbe quella di eliminare un elemento marcandolo con un apposito valore ***deleted***, tuttavia il costo computazionale della ricerca e dell'inserimento non dipenderebbero più esclusivamente dal fattore di carico ma **anche dal numero di posizioni marcate**.

  Per queste ragioni, di solito la cancellazione **non è supportata con l'indirizzamento aperto**.

Hashing doppio

Garantire l'**ipotesi di uniformità semplice** risulta essere quasi impossibile, tuttavia, alcune tecniche si avvicinano molto ad essa, in particolare l'**hashing doppio**.

L'idea alla base è quella di usare **due diverse funzioni hash**, una per determinare l'a**ccesso iniziale** alla tabella e l'altra per determinare il **passo di scansione** (ossia la ripetizione con l'incremento in base alla collisione):

$$h(k, i) = [h_1(k) + i \cdot h_2(k)] \ \texttt{mod} \ m \qquad \text{dove } i \in [0, m-1]$$

Se le due funzioni sono ben progettate, è **estremamente improbabile che due chiavi** $k_1 \neq k_2$ **producano una collisione su entrambe le funzioni hash**.