# Componenti delle Funzioni Ricorsive

Prendiamo il seguente **algoritmo di ordinamento ricorsivo**

**SORT(V)**

```
if V.length == 1
        return V
else
        L = SORT(first half of V)
        R = SORT(second half of V)
        S = MERGE(L, R)
        return S
```

Possiamo distinguere i **3 componenti fondamentali** di una funzione ricorsiva

**Casi Base**

```
if V.length == 1
        return V
else
        L = SORT(first half of V)
        R = SORT(second half of V)
        S = MERGE(L, R)
        return S
```

**Chiamate Ricorsive**

```
if V.length == 1
        return V
else
        L = SORT(first half of V)
        R = SORT(second half of V)
        S = MERGE(L, R)
        return S
```

*Vengono definite ricorsione in testa se avvengono prima del passo ricorsivo, ricorsione in coda se avvengono dopo*

**Passo Ricorsivo**

```
if V.length == 1
        return V
else
        L = SORT(first half of V)
        R = SORT(second half of V)
        S = MERGE(L, R)
        return S
```

**⚠ Regole della Ricorsione**

Le regole per il corretto funzionamento di una funzione ricorsiva sono

1. I **casi base** devono essere **risolti correttamente**
2. Qualunque sia l'input, le catene delle **chiamate ricorsive** devono essere ben fondate, ovvero devono **sempre arrivare ad un caso base**
3. Assumendo corretti i risultati delle chiamate ricorsive, il **passo ricorsivo** deve **produrre una soluzione corretta**

# Divide et Impera

*Dividi et Impera* (Dividi e Comanda) è una **tecnica di risoluzione dei problemi generale**, che si prostra perfettamente per la risoluzione di problemi ricorsivi

**Definizione**

1. Se l'istanza del problema da risolvere è troppo complicata per essere risolta direttamente, **dividila in sottoproblemi**

2. **Usa la stessa tecnica ricorsivamente** per risolvere i singoli sottoproblemi
3. **Combina le soluzioni** trovate per i sottoproblemi in una soluzione per il problema originario

# Equazioni Ricorsive

Per calcolare il **costo computazionale di una funzione ricorsiva**, si usano delle speciali equazioni ricorsive

Consideriamo la funzione

**FUN1(n)**

```
if n = 1
        return 1
else
        return FUN1(n - 1)
```

Analizziamo i costi delle singole parti

- Il caso base $n = 1$ avrà un costo $c_1$
- Per ogni $n > 1$ avremo un costo $T(n-1) + c_2$ dove $c_2$ è il costo del passo ricorsivo

Quindi otterremo un sistema

$$T(n) = \begin{cases} c_1 & n = 1 \\ T(n-1) + c_2 & n > 1 \end{cases}$$

$$\begin{aligned} & T(1) = c_1 \\ & T(2) = T(1) + c_2 = c_1 + c_2 \\ & T(3) = T(2) + c_2 = c_1 + 2c_2 \\ & \dots \\ & T(n) = c_1 + c_2(n-1) = \Theta(n) \end{aligned}$$

**Osservazione**

$c_1$ e $c_2$ sono **costanti additive** che non influiscono sull'ordine di grandezza del costo computazionale

Consideriamo ora invece

**FUN2(n)**

```
if n = 1:
        return 1
else
        return FUN2(n - 1) + FUN2(n - 1)
```

Possiamo aggiungere al sistema una nuova costante $c_3 = 2$ che corrisponderà al **numero di chiamate ricorsive** nella funzione

$$T(n) = \begin{cases} c_1 & n = 1 \\ c_3 T(n-1) + c_2 & n > 1 \end{cases}$$

$$T(n) = \frac{c_1 {c_3}^{n+1} + c_2 {c_3}^n - c_1 {c_3}^n - c_2 c_3}{c_3(c_3 - 1)}$$

Per semplificare, possiamo considerare solo

$$T(n) = \Theta({c_3}^n)$$

**Osservazione**

$c_3$ è una **costante moltiplicativa** che incide gravemente sull'ordine di grandezza del costo computazionale

# Ordini di Grandezza

Siccome abbiamo visto che **solo le costanti moltiplicative implicano sull'ordine di grandezza** del costo computazionale, possiamo semplificare l'equazione di una funzione ricorsiva utilizzando gli ordini di grandezza

**Esempio**

**FUN3(n)**

```
if n = 1
        return 1
else
        x = 1
        for i = 1 to n
                x = x + 1
        return FUN3(n - 1)
```

$$c_3 = 1$$

$$\begin{aligned} T(n) &= \begin{cases} c_1 & n = 1 \\ c_3 T(n-1) + c_2 n & n > 1 \end{cases} \\ &= \begin{cases} \Theta(1) & n = 1 \\ c_3 T(n-1) + \Theta(n) & n > 1 \end{cases} \\ &= \Theta\left(\begin{cases} 1 & n = 1 \\ c_3 T(n-1) + n & n > 1 \end{cases}\right) \end{aligned}$$

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

# Metodi di Risoluzione

Per risolvere le equazioni ricorsive, si possono usare **due metodi principali**

- Sostituzione
- Albero della Ricorsione

## Sostituzione

Per il metodo tramite sostituzione si prova a **indovinare una soluzione** per poi **confermarla tramite principio di induzione**

**Principio di Induzione**

Il principio di induzione permette di dire che se

1. La nostra affermazione è vera per $n = n_0$
2. Supponendo vera la nostra affermazione per ogni $m \mid n_0 \leq m \leq n-1$ possiamo dimostrare che è vera anche per $n$

Allora la nostra affermazione è vera $\forall n \geq n_0$

**Esempio**

$$T(n) = \begin{cases} 4 & n = 1 \\ T(n-1) + 3n & n > 1 \end{cases}$$

Ipotizziamo che la soluzione sia

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

Iniziamo dimostrando che

$$T(n) = O(n^2) \iff \exists\, k > 0 \mid T(n) \leq kn^2$$

Supponiamo che per $1 \leq m \leq n-1$ l'affermazione sia vera e dimostriamo per $n$

$$\begin{aligned} T(n) &= T(n-1) + 3n \\ &= k(n-1)^2 + 3n \\ &= kn^2 - 2kn + k + 3n \\ &= kn^2 - n(2k-3) + k \end{aligned}$$

$$\begin{aligned} kn^2 - n(2k-3) + k &\leq kn^2 \\ -n(2k-3) + k &\leq 0 \\ n(2k-3) + k &\geq 0 \\ n &\geq \frac{k}{2k-3} \end{aligned}$$

La nostra equazione è **verificata asintoticamente**

Possiamo analogamente dimostrare anche $T(n) = \Omega(n^2)$

**Controesempio**

$$T(n) = \begin{cases} 4 & n = 1 \\ T(n-1) + 3n & n > 1 \end{cases}$$

Ipotizziamo che la soluzione sia

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

Sappiamo che $T(n) = \Omega(n)$ in quanto abbiamo $3n$ + quantità positiva nell'equazione
Dobbiamo quindi dimostrare che

$$T(n) = O(n) \iff \exists\, k > 0 \mid T(n) \leq kn$$

Supponiamo che per $1 \leq m \leq n-1$ l'affermazione sia vera e dimostriamo per $n$

$$\begin{aligned} T(n) &= T(n-1) + 3n \\ &= k(n-1) + 3n \\ &= kn - k + 3n \end{aligned}$$

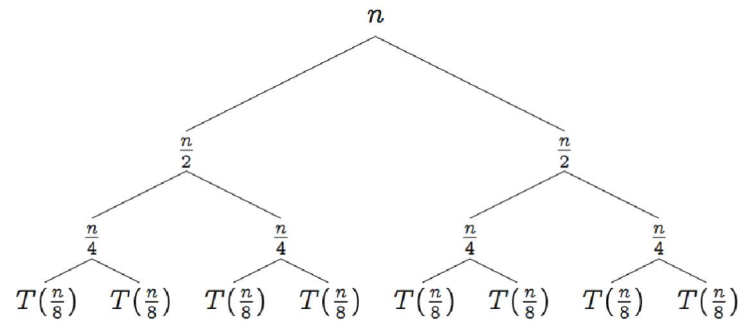
$$\begin{gathered} kn - k + 3n \leq kn \\ 3n - k \leq 0 \\ n \leq \frac{k}{3} \end{gathered}$$

L'equazione **non è verificata asintoticamente**

# Albero della Ricorsione

Prendiamo come esempio l'equazione del merge sort

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




*Albero della ricorsione della funzione Merge-Sort*

**Proprietà**

Possiamo rappresentare le chiamate ricorsive come un **albero bilanciato** dove

- La ricorsione finirà quando si arriva al piano $h$ dove si incontra $T(\text{caso base})$, quindi $h$ sarà l'altezza dell'albero
- Ogni livello dell'albero **escluse le foglie** somma a $n$
- Il livello delle foglie somma a $T(1)n$

Nel caso del merge sort

$$\frac{n}{2^{h-1}} = 1 \iff n = 2^{h-1} \iff h - 1 = \log_2(n) \iff h = \log_2(n) + 1$$
$$T(1)n = n$$

L'equazione ricorsiva quindi diventa

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

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

# Master Theorem

Metodo di risoluzione generale di

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

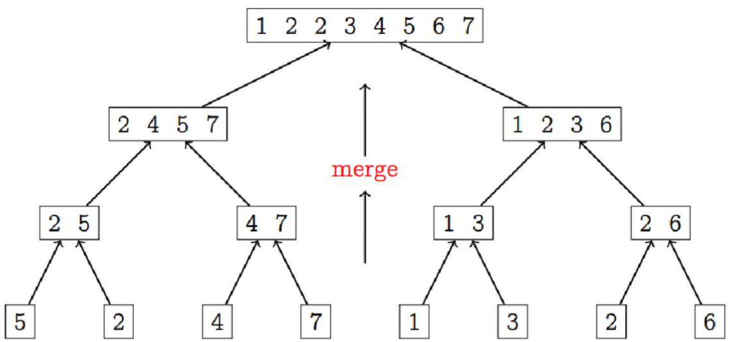
Abbiamo **3 casistiche**

1. $\exists\, \epsilon > 0 \mid f(n) = O(n^{\log_b(a)-\epsilon}) \implies T(n) = \Theta(n^{\log_b(a)})$
2. $f(n) = \Theta(n^{\log_b(a)}) \implies T(n) = \Theta(n^{\log_b(a)} \log(n))$
3. $\exists\, \epsilon > 0, c < 1, n_0 > 0 \mid$
   $\forall n \geq n_0, f(n) = \Omega(n^{\log_b(a)+\epsilon}) \land af\left(\frac{n}{b}\right) \leq cf(n)$
   $\implies T(n) = \Theta(f(n))$

# Merge Sort

**💡 Idea**

1. Se il vettore da ordinare contiene un solo elemento è già ordinato, altrimenti dividilo in due metà
2. Chiama ricorsivamente la funzione sulle due metà
3. Costruisci una soluzione fondendo insieme le due metà ordinate




Viene definito un **algoritmo bottom-up** in quanto la logica di ordinamento parte dal fondo quanto la divisione arriva ai casi base per poi ricostruire il vettore risalendo l'albero

**MERGE-SORT(A, p, r)**

```
if (p < r)
        // arrotondamento per difetto per non avere decimali
        q = FLOOR((p + r) / 2)
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

***Parametri***: *A=vettore, p=indice inizio vettore, r=indice fine vettore*

**MERGE(A, p, q, r)**

```
n1 = q - p + 1, n2 = r - q
let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
        L[i] = A[p + i - 1]
for j = 1 to n2
        R[j] = A[q + j]

L[n1 + 1] = R[n2 + 1] = infinity
i = 1, j = 1
for k = p to r
        if L[i] <= R[j]
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
```

***Parametri***: *A=vettore, p=indice inizio vettore, q=indice mediano vettore, r=indice fine vettore*

# Costo Computazionale

Possiamo calcolare che il costo computazionale della **funzione merge** è

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

L'equazione ricorsiva sarà quindi

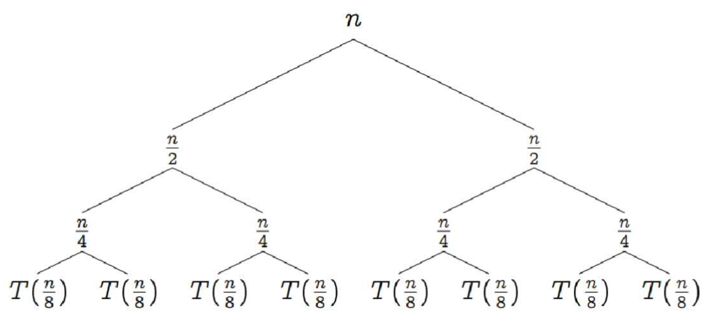
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

L'albero della ricorsione sarà quindi

## Albero della Ricorsione

Prendiamo come esempio l'equazione del merge sort

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




*Albero della ricorsione della funzione Merge-Sort*

**Proprietà**

Possiamo rappresentare le chiamate ricorsive come un **albero bilanciato** dove

- La ricorsione finirà quando si arriva al piano $h$ dove si incontra $T(\text{caso base})$, quindi $h$ sarà l'altezza dell'albero
- Ogni livello dell'albero **escluse le foglie** somma a $n$
- Il livello delle foglie somma a $T(1)n$

Nel caso del merge sort

$$\frac{n}{2^{h-1}} = 1 \iff n = 2^{h-1} \iff h - 1 = \log_2(n) \iff h = \log_2(n) + 1$$

$$T(1)n = n$$

L'equazione ricorsiva quindi diventa

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

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

Non si presentano parti dipendenti dal valore dell'input, di conseguenza **tutte e tre le casistiche saranno equivalenti**

**Costi Computazionali**

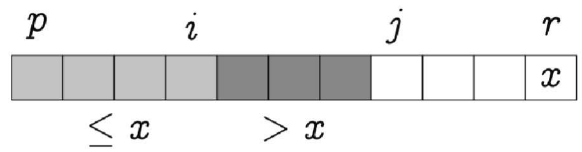
- **Caso Pessimo**: $\Theta(n\log(n))$
- **Caso Medio**: $\Theta(n\log(n))$
- **Caso Ottimo**: $\Theta(n\log(n))$

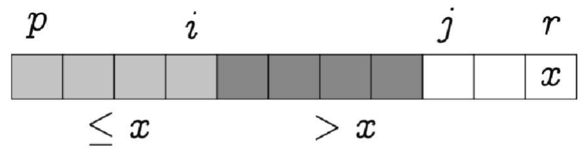
# Quick Sort

**Idea**

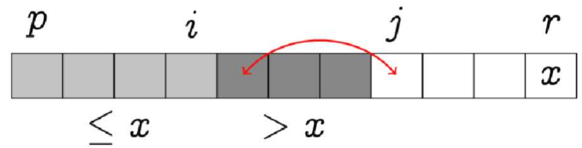
1. Se il vettore da ordinare contiene un solo elemento è già ordinato
2. Scegli un elemento di pivot e partiziona il vettore intorno al pivot
3. Chiama ricorsivamente Quick Sort sui due vettori ottenuti

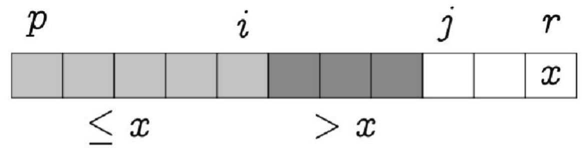
$$A[j] > x$$




$$A[j] \leq x \implies \text{scambio } A[i+1] \text{ con } A[j]$$




**PARTITION(A, p, r)**

```
x = A[r] // pivot
i = p - 1
for j = p to r
        if A[j] <= x
                i = i + 1
                exchange A[i] with A[j]
return i
```

***Parametri***: *A=vettore, p=indice inizio vettore, r=indice fine vettore*
***Return***: *posizione del pivot nel vettore a fine partizionamento*

**QUICK-SORT(A, p, r)**

```
if p < r
        q = PARTITION(A, p , r)
        QUICK-SORT(A, p, q - 1)
        QUICK-SORT(A, q + 1, r)
```

***Parametri***: *A=Vettore, p=indice inizio vettore, r=indice fine vettore*

**Osservazione: Confronto con Merge Sort**

Nonostante il caso peggiore di ordine più grande rispetto al merge sort, grazie alla tecnica di randomizzazione del pivot si ricade sempre nel caso medio, il quale ha un ordine di grandezza equivalente ma utilizzando **operazioni molto meno costose rispetto al merge**.

Per questo motivo, il Quick Sort è generalmente la **scelta preferita in campo di algoritmi di ordinamento**

# Costo Computazionale

Sapendo che il **costo computazionale del partizionamento** è

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

Possiamo avere due casi principali

- Il partizionamento avviene **perfettamente bilanciato** (A[q] era il valore mediano del vettore)

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

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

- Il partizionamento avviene **completamente sbilanciato** (A[q] era il più piccolo o più grande valore del vettore)

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

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

Per trovare l'ordine di grandezza del **caso medio** abbiamo due metodi

- Possiamo considerare che Il caso medio consiste semplicemente in **partizioni bilanciate e sbilanciate alternate**, che quindi si compensano tra di loro
  1. sbilanciata: $T(n) = T(n-1) + \Theta(n)$
  2. bilanciata: $T(n) = 2T\left(\frac{n-1}{2}\right) + \xcancel{\Theta(n) + \Theta(n)}^{\Theta(n)}$

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

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

- Possiamo considerare le partizioni come $\frac{1}{k}n$ elementi da un lato e $\frac{k-1}{k}n$ elementi dall'altro

$$k > 1$$

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T\left(\frac{1}{k}n\right) + T\left(\frac{k-1}{k}n\right) + \Theta(n) & n > 1 \end{cases}$$

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

**Osservazione: Albero Sbilanciato**

Anche se l'albero della ricorsione è sbilanciato da uno dei due lati, essendo le due funzioni sommate tra di loro l'ordine di grandezza corrisponderà al più grande tra le due, che possiamo calcolare essere $\Theta(n \log(n))$

**Costi Computazionali**

- **Caso Ottimo**: $\Theta(n \log(n))$
- **Caso Medio**: $\Theta(n \log(n))$
- **Caso Pessimo**: $\Theta(n^2)$

# Randomizzazione del Pivot

**Idea**

Per la legge dei grandi numeri, se si sceglie un **pivot casuale ogni volta che si esegue una partizione**, si cadrà sempre nel caso medio in quanto è infinitesimamente piccola la probabilità che la divisione sia sempre sbilanciata

**RANDOMIZED-QUICK-SORT(A, p, r)**

```
if p < r
        q = RANDOMIZED-PARTITION(A, p, r)
        RANDOMIZED-QUICK-SORT(A, p, q-1)
        RANDOMIZED-QUICK-SORT(A, q + 1, r)
```

***Parametri***: *A=vettore, p=indice inizio vettore, r=indice fine vettore*

**RANDOMIZED-PARTITION(A, p, r)**

```
i = RANDOM(p, r) // random pivot
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

***Parametri***: *A=vettore, p=indice inizio vettore, r=indice fine vettore*
***Return***: *posizione del pivot nel vettore a fine partizionamento*