Divide et Impera

1. **Divide**: divido il problema in sottoproblemi.
2. **Impera**: risolvo tutti i sottoproblemi

    > *sottoproblema*: problema su un istanza più piccola

3. **Combina**: combino i risultati dei sottoproblemi.

Questo approccio si realizza molto bene applicando la *ricorsione*.
Una volta che i sottoproblemi diventano sufficentemente piccoli da non richiedere ricorsione diciamo che la ricorsione ha toccato il caso base. A volte, oltre ai sottoproblemi, dobbiamo risolvere dei sottoproblemi che non sono affatto uguali al problema originale. Noi consideriamo la risoluzione di questi problemi nella fase **combina**.

Ricorrenze

Le ricorrenze vanno mano nella mano con il divide et impera. Una ***ricorrenza*** è un equazione o disequazione che descrive una funzionein termini del suo valore con input più piccoli.
Le ricorrenze possono assumere varie forme. Per esempio un alg. può suddividere i sottoproblemi in dimensioni differenti.

> *Esempio*:
> $n! = n(n-1)(n-2)...2 \cdot 1$

Risolvere le equazioni di ricorrenza

Le equazioni di ricorrenza vengono utilizzate per valutare il costo computazionale degli algoritmi ricorsivi.
La prima cosa da fare è quella di scrivere il costo computazionale sotto forma di equazione di ricorrenza.
Ci sono tre metodi principali per risolvere le equazioni di ricorrenza:

- Master Theorme
- Metodo Iterativo
- Metodo dell'Albero di Ricorsione

posso dare questa definizione in modalità ricorsiva:

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

Pseudo codice:

```
fatt(n)
   if n = 1     //caso base
      then return 1
      else return n * fatt(n - 1)
```

ogni volta che la funzione richiama se stessa, l'esecuzione della funzione chiamante viene sospesa, e così va avanti a cascata fino all'arrivo di un caso base.
Notiamo bene però che dal punto di vista pratico la ricorsione è molto pesante, infatti il sistema deve tenere in memoria lo stato di ogni funzione chiamante.

Binary search (con ricorsione)

```
Binary_Search(A, i, j, x)
   if j - i < 0
      then return - 1
      else k = int_floor((i + j) / 2)

      if A[k] = x
         then return k
      if A [k] < x
         then return Binary_Search(A, k + 1, j, x)
         else return Binary_Search(A, i, k - 1, x)
```

Numeri di Fibonacci

La ricorsione spesso ci può portare a calcolare la stessa cosa più volte

$$fib(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n \geq 2 \end{cases}$$

```
fib(n)
   if n = 0 then return 1
   if n = 1 then return 1
      else return fib(n-1)+fib(n-2)
```

Stessi valori vengono calcolati più e più volte.
$C(n) = \text{ n}^\circ \text{chiamate ricorsive } n > 1$
$C(n) \leq 2(C(n-1)) \leq 2 \cdot 2C(n-2) \leq ... \leq 2^i \cdot C(n-i) \leq ... \leq 2^{n-2} \cdot C(n-(n-2)) = 2^{n-1}$

$C(n) \in O(2^n)$ BRUTTO!

Le soluzioni ricorsive non sempre quindi sono delle buone idee (come in questo caso).

Nel caso in cui il bit dipende da 1 valore la dim. dell'input dipende dai bit per rappresentare n (log n bit)

$$logn \xRightarrow{\text{elevo potenza}} O(n)$$

Tutti gli algoritmi che prendono un valore in input e sono di ordine O(n) hanno bisogno di un numero esponenziale nella dimensione dell'input.

Torre di Hanoi

Scopo: Trasportare la priamide da 1 a 3 mantenendo l'ordine:

1. un disco alla volta
2. non posso mettere un disco più grande sopra uno più piccolo

```
Hanoi(m, s, d, a)
   if m = 1 then muovidisco(s, d)
            else Hanoi(m - 1, s, a, d)
               muovidisco(s, d)
               Hanoi(m - 1, a, d, s)
```

Costo Computazionale

Quante mosse devo fare?

$$\begin{cases} M(1) = 1 \\ M(m) = 1 + 2M(m-1) \end{cases}$$

Vogliamo trovare una funzione che possiamo calcolare partendo da

$$
\begin{aligned}
M &= 1 + 2M(m-1) \\
&= 1 + 2M(1 + 2M(m-2)) \\
&= 1 + 2 + 4 + 8(1 + 2M(m-4)) \\
&= \dots \\
&= \sum_{J=0}^{i+2} 2^J + 2^{i-1}(1 + 2M(m-i)) \\
\hookrightarrow &= \sum_{J=0}^{m-3} 2^J + 2^{m-2}(1 + \underbrace{2M(1)}_{=1}) \\
&= \sum_{J=0}^{m-3} 2^J + 2^{m-2} + 2m - 1 \\
&= \sum_{J=0}^{m-1} 2^j = 2^m - 1
\end{aligned}
$$

Manca solo dimostrare per induzione che $M(m) = 2^m - 1$

Caso base: $M(1) = 2^1 - 1 = 1$ OK!

Ipotesi induttiva: $M(m-1) = 2^{m-1} - 1 \backslash$

Passo induttivo:

$$
\begin{aligned}
M(m) &= 1 + 2M(m-1) \\
&= 1 + 2(\underbrace{2^{m-1} - 1}_{\text{IP}}) = 1 + 2^m - 2 = 2^m - 1
\end{aligned}
$$

Quindi il conto fatto prima era giusto.

Notiamo che anche se è esponenziale il nostro algoritmo è ottimo!

L'over bound ***al problema*** è esponenziale.

Equazioni di ricorrenza

```
Algric(I)
   if I "è piccolo"
      then return output
      else
         Algric(I1)
         Algric(I2)
```

> I input chiamata esterna
> I_1 e I_2 input delle chiamate ricorsive

A volte è possibile ricondurre il nostro costo computazionale ad un teorema ben preciso.

Come posso scrivere il costo computazionale di questo algoritmo?

$n = n_0$ caso base

$$T(n) = \begin{cases} T(n_0) \\ T(n_1) + T(n_2) + ... + T()n_a + g(n) & n > n_0 \end{cases}$$

Non è detto però che le chiamate ricorsive siano fatte su sottoproblemi tutti della stessa dimensione, ma quando succede, ovvero quando: $n_1 = n_2 = ... = n_a = \frac{n}{b}$
Possiamo utilizzare il:

Master Theorem (teorema dell'esperto)

Se:

$$T(n) = \begin{cases} \Theta(1) & n = n_0 \\ a \cdot T(\frac{n}{b}) + O(n^d) & n > n_0 \end{cases}$$

$$\text{con } a > 0;\ b > 1;\ d \geq 0$$

Allora:

$$T(n) = \begin{cases} O(n^d) & \text{se } d > log_b a \\ O(n^d log n) & \text{se } d = log_b a \\ O(n^{log_b a}) & \text{se } d < log_b a \end{cases}$$

Merge Sort

Ordinamento
Input: A = A[0]...A[n-1]
Output: voglio A ordinato in senso non decrescente ovvero: A[i] $\leq$ A[i+1]
un Sottoproblema può essere ordinare una sequenza più piccola.

L'algortimo **Merge Sort** si basa proprio sulla tecnica Divide et Impera.
Divide: divido A in 2 parti
Impera: Ordino le due parti in maniera indipendente una dall'altra
Combina: merge delle due parti

Ordino le due metà in maniera ricorsiva ovvero come il problema più grande.
Continuo a chiamare la mia ricorsione fino a quando non mi rimane un solo elemento.

```
Mergesort(A, i, j)
   if i < j
      then k = ⌊(i+j)/2⌋
           Mergesort(A, i, k)
           Mergesort(A, k+1, j)
           Merge(A, i, k, j)
```

Esempio:

Merge

Input: A= $_i|\rule{3cm}{0.4pt}|_k\ |\rule{3cm}{0.4pt}|_j$

Output: A ordinato

```
Merge(A, i, k, j)
   n1 = k-i+1
   n2 = j-k
   crea L[0...n1] e R[0...n2]
   for t=0 to n1-1
      L[t]:=A[i+t]
   for t=0 to n2-1
      R[t]:=A[k+1+t]
   L[n1]:= ∞
   R[n2]:= ∞
   l = 0
   r = 0
   for t = i to j
      if L[l] ≤ R[r]
         then A[t]:=L[l]
              l:=l+1
         else A[t] := R[r]
              r:=r+1
```

T(n) di Mergesort

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

a = 2

a = 2

d = 1

$log_b a = log_2 2 = 1 \Rightarrow T(n) \in O(nlogn)$

$$Size = n$$
$$c \cdot n$$