Limiti inferiori e algoritmi ottimi

corso di laurea in **Matematica**

Informatica Generale, Lezione **7(b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Limiti inferiori

Abbiamo visto 3 algoritmi che ordinano un vettore in tempo $\Theta(n^2)$.
Sorge spontanea la domanda: è il **meglio che si può fare**?

Le possibili risposte sono:

- **NO**: in tal caso è sufficiente **esibire un algoritmo** che ha una complessità **migliore** (sotto le stesse assunzioni)
- **SI**: in tal caso è necessario **dimostrare** che **nessun algoritmo** risolve il problema **con meno operazioni**.

Nel secondo caso, stabiliamo un **limite inferiore** al **problema**. Un **qualsiasi algoritmo**, invece, stabilisce un **limite superiore**.

Se conosciamo un algoritmo $\mathcal{A}$ che ha **complessità uguale al limite inferiore** del problema che risolve, diremo che $\mathcal{A}$ è **ottimo**.

Dovrebbe essere intuitivo che calcolare i limiti inferiori "stretti" in generale è difficile, e spesso trattasi problemi aperti.

Ordinamento: spazio delle soluzioni

Vedremo di stabilire un **limite inferiore** dell'**ordinamento** e poi vedremo un **algoritmo ottimo**. Cos'è il risultato di ordinamento?

Una **permutazione** v' di un vettore v, che soddisfa la proprietà di essere ordinato, **Asc**(v').

Quindi, in un certo senso, un algoritmo di ordinamento individua l'**unica permutazione ordinata** tra le $n!$ permutazioni di un vettore con n elementi.

Quali strumenti usano gli algoritmi visti finora per "scegliere" la permutazione? Essenzialmente **confrontano** gli elementi e sulla base del risultato (eventualmente) **spostano** degli elementi. Gli algoritmi visti finora e i prossimi, sono detti **basati su confronti**.

Possiamo quindi immaginare un **algoritmo di ordinamento** come una **ricerca** di un elemento **nello spazio** di tutte le possibili **soluzioni**, che sono le **permutazioni** di un vettore.

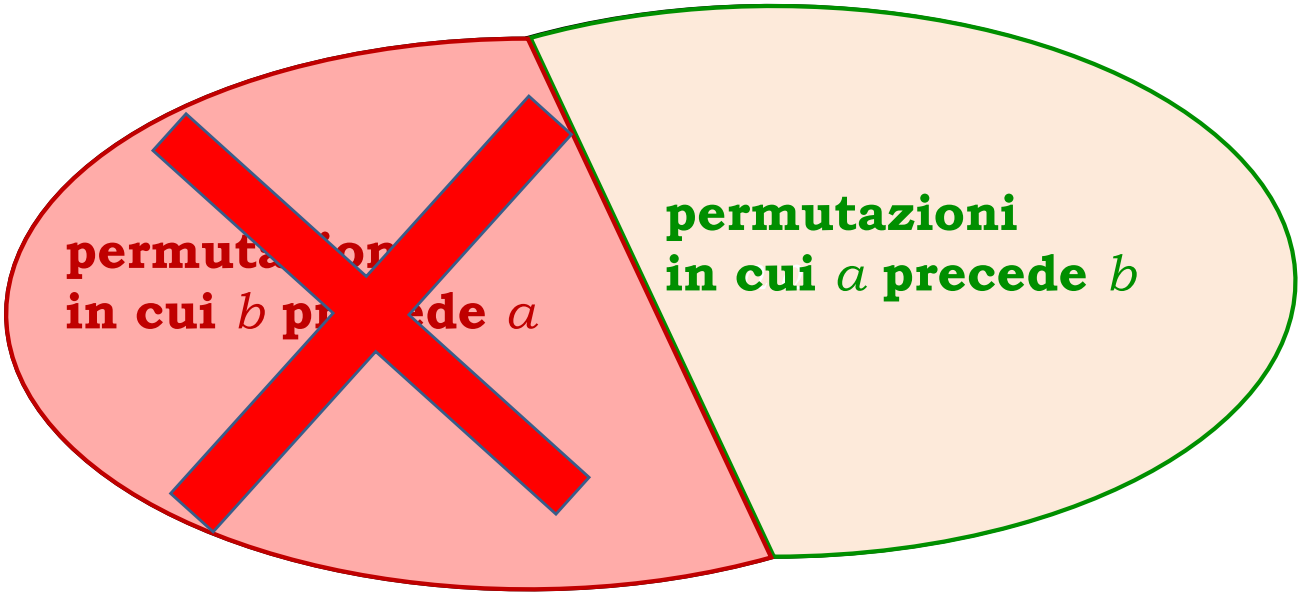
Che informazione mi dà un confronto

Domanda: Qual è il **massimo progresso** che possiamo sperare di ottenere confrontando due valori a e b in un vettore?

Risposta: Sapendo $a < b$ possiamo **dividere in due** lo spazio delle soluzioni: le **permutazioni buone B** in cui ***a* viene prima di *b***

e le **permutazioni cattive C**, in cui il **valore *a* viene dopo di *b***.

A quel punto possiamo immaginare di **continuare la ricerca su B**.

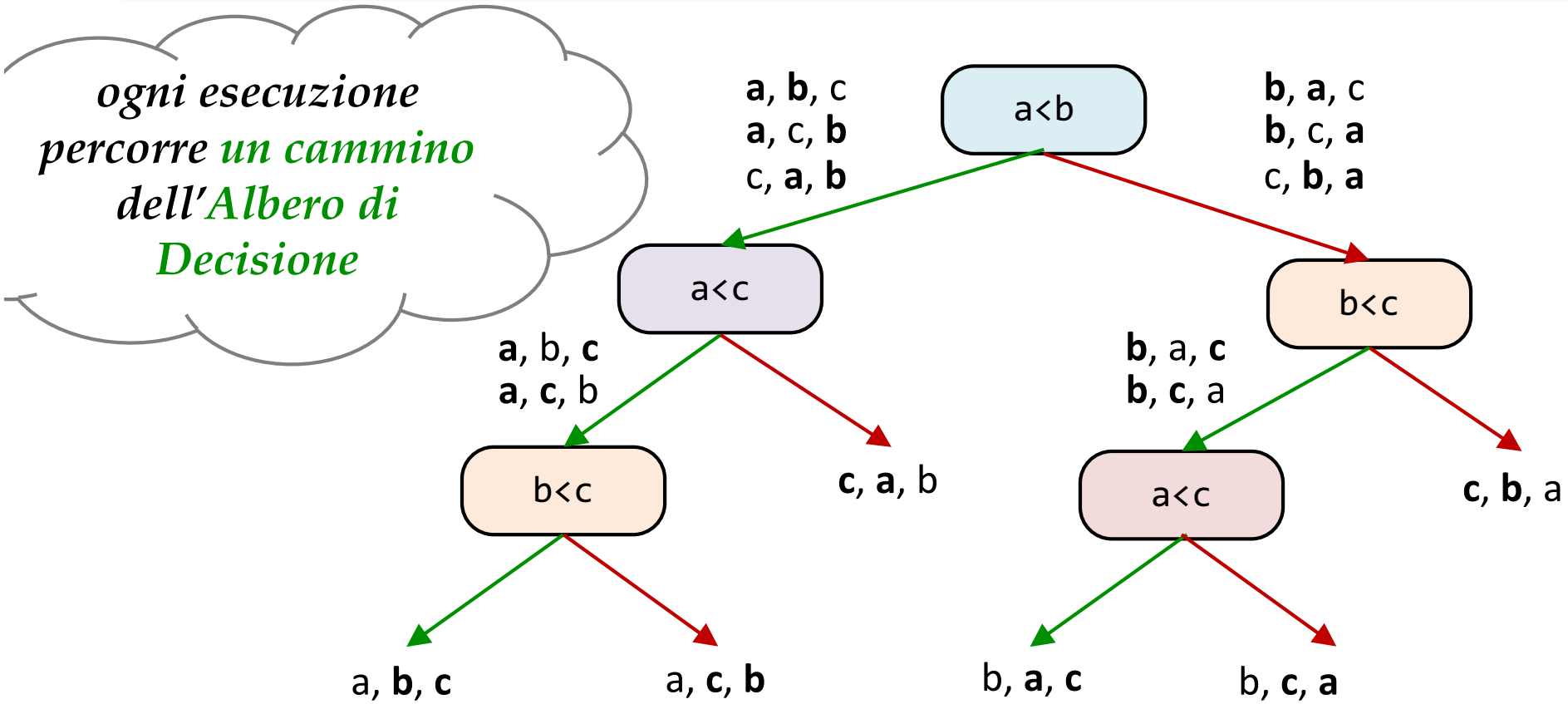


Esempio: ordinamento di 3 numeri

Vediamo come un ordinamento potrebbe "cercare" la permutazione giusta tra tutte le 3! = 6 permutazioni di tre valori.

Albero di decisione dell'algoritmo di massimo di 3 numeri, che è chiaramente ottimo. Assomiglia a una **ricerca binaria**.

ogni esecuzione percorre un cammino dell'Albero di Decisione



Limite inferiore per l'ordinamento

Come nella ricerca binaria, un **algoritmo ottimo**, almeno in **linea di principio**, potrebbe percorrere **un solo cammino** nell'albero delle scelte e dirigersi senza indugi verso la **permutazione ordinata**.

Oltre alla ricerca binaria, può essere utile ricordare che l'**altezza** di un **albero binario bilanciato** è **logaritmica** rispetto al numero delle **foglie**. Di conseguenza, possiamo dire che il problema è $\theta(\log_2 n!)$.

Abbiamo già visto che $\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$ e applicando i logaritmi questo implica: $\log n! = \boldsymbol{\Omega}(n \log n)$. Abbiamo dimostrato il seguente:

- **Teorema**: *Ogni algoritmo di ordinamento* ***basato su confronti*** *è* $\boldsymbol{\Omega}(n \log n)$.

La complessità degli algoritmi di ordinamento che abbiamo visto è $\theta(n^2)$ quindi **maggiore del limite inferiore** che abbiamo appena stabilito per l'ordinamento $\boldsymbol{\Omega}(n \log n)$.

Nel seguito vedremo un **algoritmo ottimo**.

Riflessioni generali

L'**ordinamento** pare essere un problema **più che lineare**:

Supponiamo di avere un algoritmo $\mathcal{A}$ che si risolve un certo problema in esattamente an^2 passi per una certa costante a.

Supponiamo che sia possibile suddividere risolvere il problema in $k > 1$ sotto-istanze e che combinare i risultati costi bn (b costante).

Dividiamo un'istanza in k sotto-istanze e poi le ricombiniamo. Come cambia la complessità?

$$k\ a\left(\frac{n}{k}\right)^2 + bn = a\,\frac{n^2}{k} + bn$$

facciamo un progresso per **la costante moltiplicativa** k (per n sufficientemente grandi, quando il $n^2 >> n$).

Ma se ripetiamo il processo finché possibile, fino a raggiungere istanze del problema **elementari** in tempo costante?

Forse la ricorsione può aiutarci….

Divide et impera

Supponiamo di riuscire a **suddividere** l'istanza I di un problema in k **sotto-istanze** $I_1, I_2, \ldots, I_k$ ($k > 1$ costante fissata, spesso $k = 2$) e di riuscire a calcolare "**facilmente**" la soluzione s di I, a partire dalle soluzioni $s_1, s_2, \ldots, s_n$ delle istanze $I_1, I_2, \ldots, I_n$.

Potremo pensare alla seguente procedura ricorsiva.

Ovviamente il **successo** di questa strategia **dipende** dal **costo di divisione** e **ricombinazione** rispetto al costo del problema originale.

```
def risolvi(I):
    if elementare(I):  # caso base/elementare
        return risolviElementare(I)
    I_1, … , I_k = dividi(I)              # dividi
    forall i ∈ [1,k] :
            s_i = risolvi(I_i)            # impera
    s = ricombina(s_1, … , s_k)           # ricombina
    return s
```

Ordinamento Div&Imp: mergeSort

[John **von Neumann**, **1945**, pubblicato **1948**]

(**1**) **Istanze elementari**: un vettore lungo **1** oppure **0** è **già ordinato**.

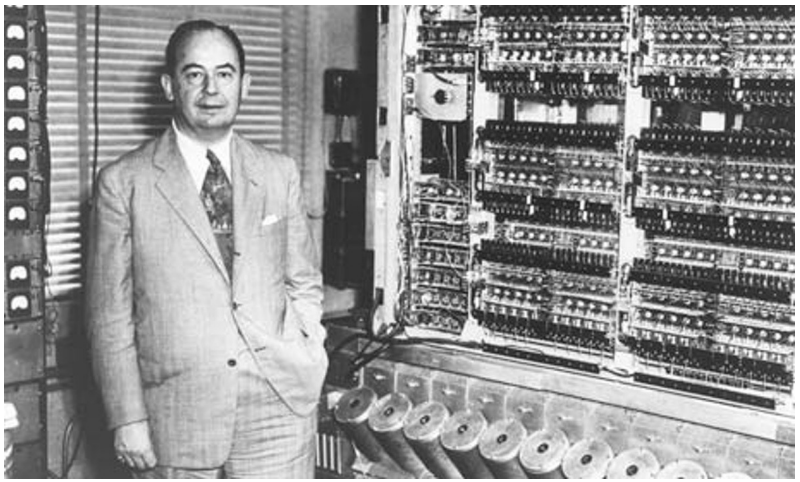
(**2**) **Divisione**: possiamo banalmente dividere un vettore *v*[*inf*, *sup*) in due metà v[*inf*, *m*) e v[*m*, *sup*) semplicemente calcolando il punto medio *m* tra *inf* e *sup*.

(**3**) **Soluzione sottoistanze**: lo so fare per **ipotesi induttiva** ☺

(**4**) **Ricombinazione** (**Impera**): dati due vettori ordinati, quanto è difficile produrre un vettore ordinato con gli elementi di entrambi?

L'attività (**4**) è l'unica non banale, ma dovrebbe ricordarvi un problema analogo, il problema dei Punti Coincidenti.

```
def mergeSort(v, inf, sup):
    if sup-inf > 2:
    # v[inf, sup) ha almeno 2 elem.
        m = puntoMedio(inf, sup)
        mergeSort(v, inf, m)
        mergeSort(v, m, sup)
        fondi(v, inf, m, sup)
```



Fusione ordinata: specifiche

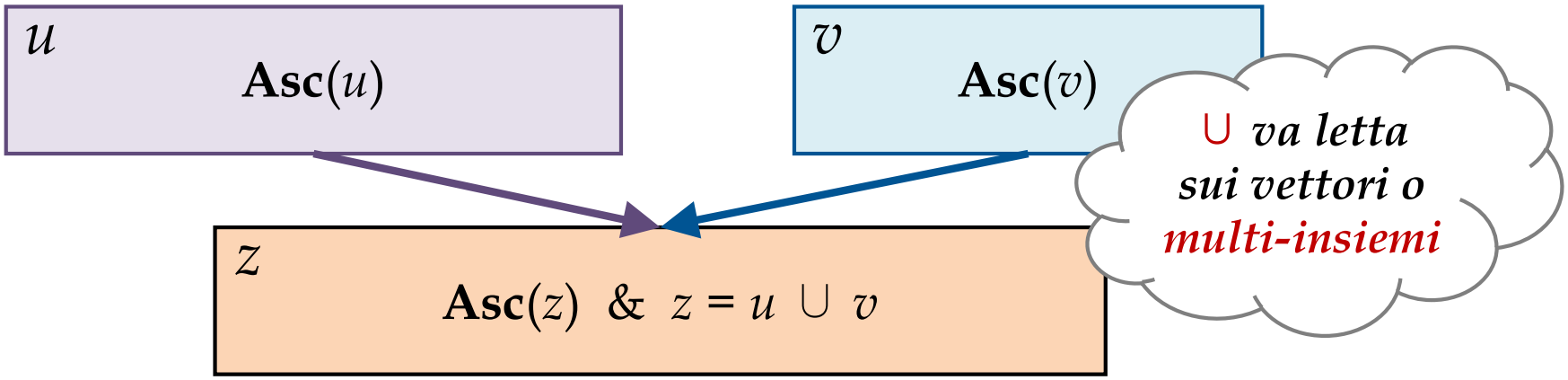
Come spesso accade, è conveniente scrivere una funzione più generica, che **fonde due vettori ordinati** u e v in un terzo **vettore ordinato** $z[0, p)$ con $p = m+n$.

Le **precondizioni** sono **Asc**$(u[0, m))$ & **Asc**$(v[0, n))$…

e vogliamo assicurare la **post-condizione**:

$$\textbf{Asc}(z[0, p)) \;\&\; z[0, p) = u[0, m) \cup v[0, n) \;\&\; p=m+n$$

Questo problema presenta profonde **analogie** con il problema di calcolare il **numero dei punti coincidenti** (caso ordinato) con una principale differenza: quando il vettore u (risp. v) termina prima di v (risp. u) **occorre ricopiare** quanto rimane di v (risp. u) in z.



Fusione ordinata: asserzioni

Indebolimento: assumendo un **processo iterativo** di scorrimento dei vettori u e v, e introducendo **3 variabili** i, j e k per scorrere i 3 vettori, a una generica iterazione vogliamo soddisfare la condizione:

$\varphi(i, j, k) \equiv z[0, k) = u[0, i) \cup v[0, j)$ (k elementi in z presi da u e v)

& **Asc**$(z[0, k))$ (in z sono **ordinati**)

& $z[0, k) \leq u[i, m) \cup v[j, m)$ (e sono **minori dei rimanenti**)

dove $i < m$ e $j < n$ e di conseguenza anche $i + j = k < p = m+n$.

$\varphi(i, j, k)$ implica l'asserzione desiderata per $i = m$ & $j = n$ (necessariamente da $z[0, k) = u[0, i) \cup v[0, j)$ ho anche $k = i + j$)

…ed è banalmente verificata per $i = j = k = 0$.

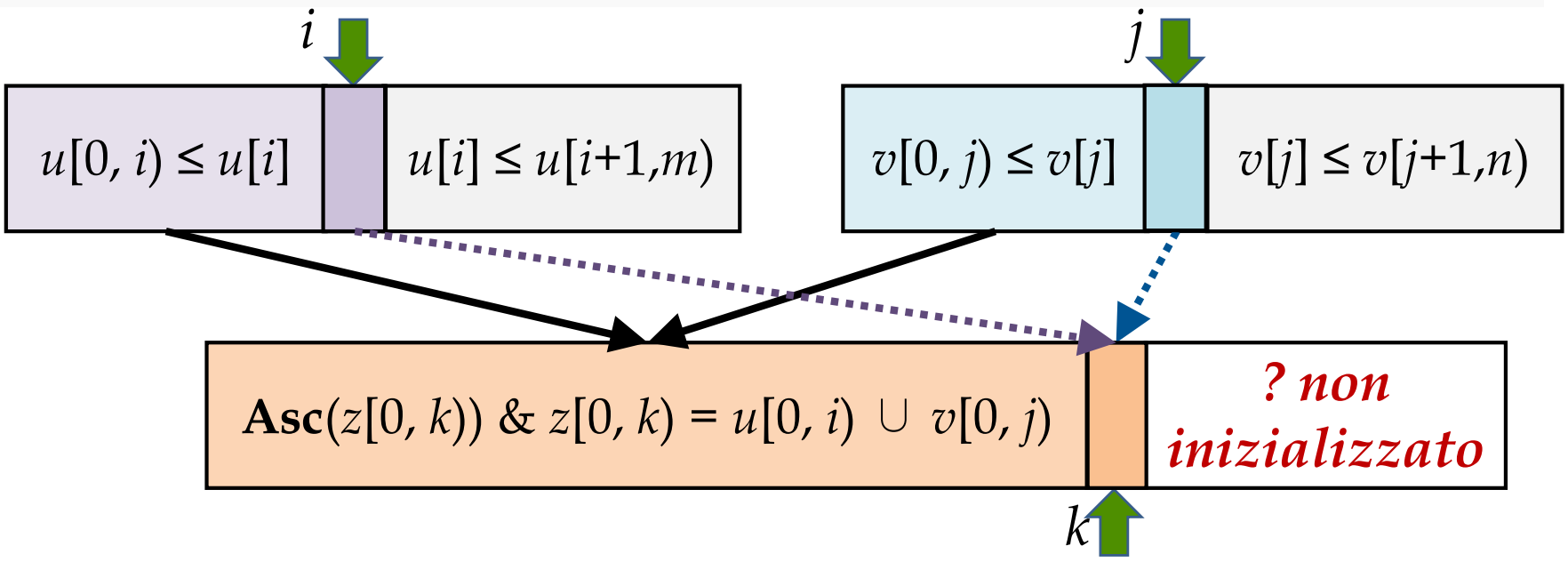
Abbiamo quindi le **inizializzazioni** e una **possibile guardia** (ma occorre capire cosa fare quando si finisce di copiare u o v)

Funzione merge: passo generico

Confronto $u[i]$ e $v[j]$:

- se $u[i] \leq v[j]$ allora $u[i]$ è anche **min**$(u[i,m) \cup v[j, n))$ e ricordando $u[i] \geq z[0, k)$ lo ricopio in z, assicurando **Asc**$(z[0, k+1))$, e quindi avanzo su u (incrementando i) e su z (incrementando k)
- se $u[i] \geq v[j]$ allora $v[j]$ è anche **min**$(u[i,m) \cup v[j, n))$ e ricordando $v[j] \geq z[0, k)$ lo ricopio in z e avanzo su v (incrementando j) e su z (incrementando k)

Quindi il min{ $u[i]$, v[j] } è il **minimo dei non ricopiati** di u (risp. v).



Fusione ordinata: pseudocodice

L'ultima osservazione è che le operazioni descritte prima si possono fare finché entrambi i vettori non sono finiti, cioè finché $i < sup$ e $j < inf$, **altrimenti** il confronto $u[i] \leq v[j]$ **non ha senso**.

Dopodiché è sufficiente **ricopiare la coda** del vettore non terminato.

➧ **Esercizio**: scrivere *merge* in modo che non sia necessario "ricopiare le code" (occorre fare tutto nel ciclo principale – **attenzione alle condizioni**!). **C'è vantaggio?**

```
def merge(u, infu, supu, v infv, supv, z, infz):
    i, j, k = infu, infv, infz
    while i < supu and j < supv:
        if u[i] ≤ v[j]: z[k], i = u[i], i+1
                   else: z[k], j = v[j], j+1
        k = k + 1
    # F: i = supu or j = supv
    while i < supu: z[k], k, i = u[i], k+1, i+1
    while j < supv: z[k], k, j = v[j], k+1, j+1
```

si entra sempre in uno solo dei due while finali

Analisi di mergeSort

Analizziamo la complessità dell'algoritmo **ricorsivo**, versione base, tanto gli altri **non migliorano la complessità asintotica** (anche se possono essere **molto più efficienti**).

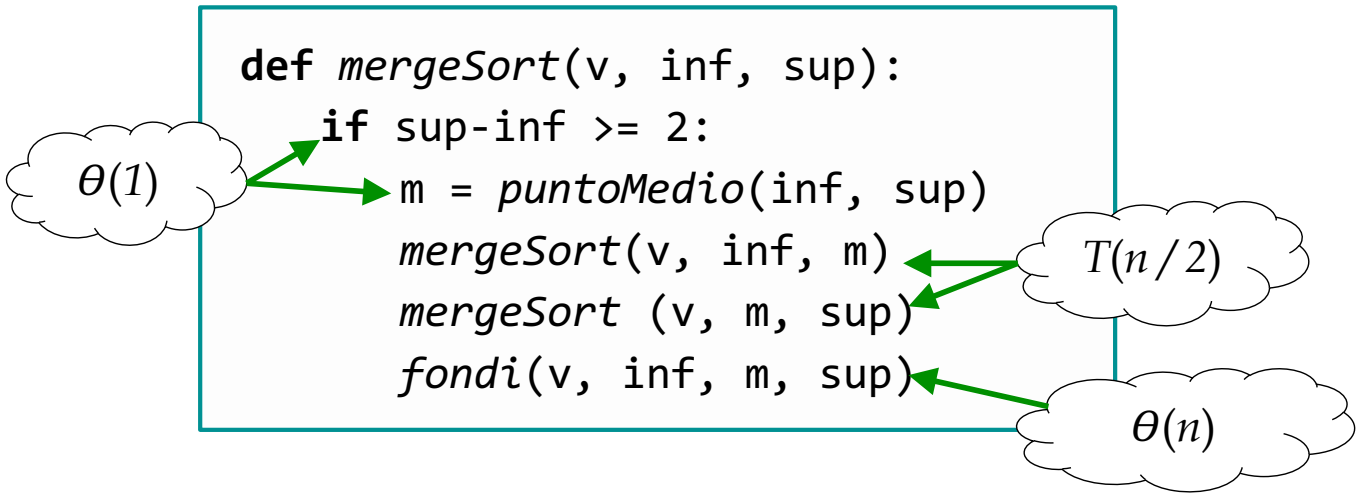
Divisione e caso base sono chiaramente costanti.

Sappiamo che la **fusione è lineare** e chiamando $n = sup - inf$…

Infine, detta $T(n)$ la complessità di `mergeSort`, le due **chiamate ricorsive** avranno ovviamente complessità $T(n/2)$.

Abbiamo quindi la **relazione di ricorrenza**:

$$T(n) = 2T(n/2) + \Theta(n) \qquad T(1) = \Theta(1)$$



Review: metodo iterativo

Risolviamo l'equazione con il **metodo iterativo**:

$$T(n) = 2\,T(n/2) + \theta(n) =$$
$$= 2[2T(n/2^2) + \theta(n/2^1)] + \theta(n/2^0) =$$
$$= 2[2[2T(n/2^3) + \theta(n/2^2)] + \theta(n/2^1)] + \theta(n/2^0) =$$
$$= 2^3T(n/2^3) + 2^2\theta(n/2^2) + 2^1\,\theta(n/2^1) + 2^0\,\theta(n/2^0) =$$

… (si "**srotola**" fino a una **forma generale**)

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

fino a $2^k = n$, cioè $k = \log_2 n$. Sapendo che $2^{\log n} = n$ ottenendo:

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

"pericolosi" movimenti di costanti dentro e fuori θ!!!