

Algoritmi e Strutture Dati

Analisi di algoritmi Introduzione

Alberto Montresor and Davide Rossi

Università di Bologna

30 settembre 2024

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

- ① Modelli di calcolo
 - Definizioni
 - Esempi di analisi
 - Ordini di costo
- ② Notazione asintotica
 - Definizioni
 - Esercizi
- ③ Costo problemi vs algoritmi
 - Sommare numeri binari
 - Moltiplicare numeri binari
- ④ Tipologia dell'input
 - Selection Sort
 - Insertion Sort
 - Merge Sort

Introduzione

Obiettivo: **stimare il costo in tempo**

- Definizioni
- Modelli di calcolo
- Esempi di valutazioni
- Ordini di costo

Perché?

- Per stimare il tempo impiegato per un dato input
- Per stimare il più grande input gestibile in tempi ragionevoli
- Per confrontare l'efficienza di algoritmi diversi
- Per ottimizzare le parti più importanti

Costo

Costo: "Dimensione dell'input" → "Tempo"

- Come definire la dimensione dell'input?
- Come misurare il tempo?

Dimensione dell'input

Criterio di costo logaritmico

- *La taglia dell'input è il numero di bit necessari per rappresentarlo*
- Esempio: moltiplicazione di numeri binari lunghi n bit

Criterio di costo uniforme

- *La taglia dell'input è il numero di elementi di cui è costituito*
- Esempio: ricerca minimo in un vettore di n elementi

In molti casi...

- Possiamo assumere che gli "elementi" siano rappresentati da un numero costante di bit
- Le due misure coincidono a meno di una costante moltiplicativa

Definizione di tempo

Tempo \equiv n. istruzioni elementari

Un'istruzione si considera elementare se può essere eseguita in tempo "costante" dal processore.

Operazioni elementari

- $a *= 2$?
- $\text{Math.cos}(d)$?
- $\min(A, n)$?

Modelli di calcolo

Modello di calcolo

Rappresentazione astratta di un calcolatore

- **Astrazione**: deve permettere di nascondere i dettagli
- **Realismo**: deve riflettere la situazione reale
- **Potenza matematica**: deve permettere di trarre conclusioni "formali" sul costo

Modelli di calcolo – Wikipedia

Pages in category "Models of computation"

The following 108 pages are in this category, out of 108 total. This list may not reflect recent changes (learn more).

E cont.

- Model of computation

A

- Abstract Job Object
- Abstract machine
- Abstract state machines
- Agent-based model
- Algorithm characterizations
- Alternating Turing machine
- Applicative computing systems

F

- Finite state machine with datapath
- Finite state transducer
- Finite-state machine
- FRACTRAN
- Funnelsort

B

H

P cont.

- Probabilistic Turing machine
- Pushdown automaton

Q

- Quantum capacity
- Quantum circuit
- Quantum computer

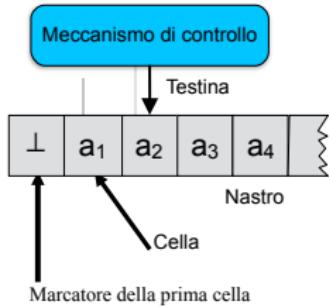
R

- Realization (systems)
- Register machine

Modelli di calcolo

Macchina di Turing

Una macchina ideale che manipola i dati contenuti su un nastro di lunghezza infinita, secondo un insieme prefissato di regole.



Ad ogni passo, la Macchina di Turing:

- legge il simbolo sotto la testina
- modifica il proprio stato interno
- scrive un nuovo simbolo nella cella
- muove la testina a destra o a sinistra

- Fondamentale nello studio della calcolabilità
- Livello troppo basso per i nostri scopi

Modelli di calcolo

Random Access Machine (RAM)

- **Memoria:**
 - Quantità infinita di celle di dimensione finita
 - Accesso in tempo costante (indipendente dalla posizione)
- **Processore (singolo)**
 - Set di istruzioni elementari simile a quelli reali:
 - somme, sottrazioni, moltiplicazioni, operazioni logiche, etc.
 - istruzioni di controllo (salti, salti condizionati)
- **Costo delle istruzioni elementari**
 - Uniforme, ininfluente ai fini della valutazione (come vedremo)

Tempo di calcolo min()

- Ogni istruzione richiede un tempo costante per essere eseguita
- La costante è potenzialmente diversa da istruzione a istruzione
- Ogni istruzione viene eseguita un certo # di volte, dipendente da n

ITEM **min**(ITEM[] A , int n)

	Costo	# Volte
ITEM $min = A[1]$	c_1	1
for $i = 2$ to n do	c_2	n
if $A[i] < min$ then	c_3	$n - 1$
$min = A[i]$	c_4	$n - 1$
return min	c_5	1

$$\begin{aligned}
 T(n) &= c_1 + c_2n + c_3(n - 1) + c_4(n - 1) + c_5 \\
 &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) = \textcolor{red}{an + b}
 \end{aligned}$$

Tempo di calcolo binarySearch()

Il vettore viene suddiviso in due parti:

Parte SX: $\lfloor (n - 1)/2 \rfloor$
 Parte DX: $\lfloor n/2 \rfloor$

int binarySearch(**ITEM**[] *A*, **ITEM** *v*, **int** *i*, **int** *j*)

	Costo	# (<i>i</i> > <i>j</i>)	# (<i>i</i> ≤ <i>j</i>)
if <i>i</i> > <i>j</i> then	c_1	1	1
return 0	c_2	1	0
else			
int <i>m</i> = $\lfloor (i + j)/2 \rfloor$	c_3	0	1
if <i>A</i> [<i>m</i>] = <i>v</i> then	c_4	0	1
return <i>m</i>	c_5	0	0
else if <i>A</i> [<i>m</i>] < <i>v</i> then	c_6	0	1
return binarySearch(<i>A</i> , <i>v</i> , <i>m</i> + 1, <i>j</i>)	$c_7 + T(\lfloor n/2 \rfloor)$	0	0/1
else			
return binarySearch(<i>A</i> , <i>v</i> , <i>i</i> , <i>m</i> - 1)	$c_7 + T(\lfloor (n - 1)/2 \rfloor)$	0	1/0

Tempo di calcolo binarySearch()

- **Assunzioni** (Caso pessimo):

- Per semplicità, assumiamo n potenza di 2: $n = 2^k$
- L'elemento cercato non è presente
- Ad ogni passo, scegliamo sempre la parte DX di dimensione $n/2$

- **Due casi:**

$$i > j \quad (n = 0) \quad T(n) = c_1 + c_2 = c$$

$$\begin{aligned} i \leq j \quad (n > 0) \quad T(n) &= T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7 \\ &= T(n/2) + d \end{aligned}$$

- **Relazione di ricorrenza:**

$$\mathcal{R} \quad T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n \geq 1 \end{cases}$$

Tempo di calcolo binarySearch()

Soluzione della relazione di ricorrenza tramite espansione

$$\begin{aligned}T(n) &= T(n/2) + d \\&= T(n/4) + 2d \\&= T(n/8) + 3d\end{aligned}$$

...

$$\begin{aligned}&= T(1) + kd \\&= T(0) + (k + 1)d \\&= kd + (c + d) \\&= d \log n + e.\end{aligned}$$

Sì risolve bene col teorema master (secondo caso)

$$n = 2^k \Rightarrow k = \log n$$

Ordini di costo

Per ora, abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due *funzioni di costo*:

- Ricerca: $T(n) = d \log n + e$
- Minimo: $T(n) = an + b$

Ordini di costo

Per ora, abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due *funzioni di costo*:

- Ricerca: $T(n) = d \log n + e$ logaritmica $O(\log n)$
- Minimo: $T(n) = an + b$ lineare $O(n)$

Ordini di costo

Per ora, abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due *funzioni di costo*:

- Ricerca: $T(n) = d \log n + e$ logaritmica $O(\log n)$
- Minimo: $T(n) = an + b$ lineare $O(n)$

Una terza funzione deriva dall'*algoritmo naïf* per il minimo:

- Minimo: $T(n) = fn^2 + gn + h$ quadratica $O(n^2)$

Ordini di costo - classi di costo asintotiche

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	logaritmico
\sqrt{n}	3	10	31	100	sublineare
n	10	100	1000	10000	lineare
$n \log n$	30	664	9965	132877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^{12}	cubico
2^n	1024	10^{30}	10^{300}	10^{3000}	esponenziale

Come sbagliare completamente l'algoritmo di controllo degli update in Windows XP e renderlo esponenziale:

<http://m.slashdot.org/story/195683>

Algoritmi e Strutture Dati

Analisi di algoritmi
Funzioni di costo, notazione asintotica

Alberto Montresor and Davide Rossi

Università di Bologna

30 settembre 2024

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

1 Modelli di calcolo

- Definizioni
- Esempi di analisi
- Ordini di costo

2 Notazione asintotica

- Definizioni
- Esercizi

3 Costo problemi vs algoritmi

- Sommare numeri binari
- Moltiplicare numeri binari

4 Tipologia dell'input

- Selection Sort
- Insertion Sort
- Merge Sort

Notazioni O , Ω , Θ

Definizione – Notazione O

Sia $g(n)$ una funzione di costo; indichiamo con $O(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \leq cg(n), \forall n \geq m$$

- Come si legge: $f(n)$ è “O grande” (big-O) di $g(n)$
- Come si scrive: $f(n) = O(g(n))$
- $g(n)$ è un **limite asintotico superiore** per $f(n)$
- $f(n)$ cresce al più come $g(n)$

Una funzione non cresce più velocemente di un certo tasso di crescita, determinato dal termine di ordine superiore

$$7n^3 + 100n^2 + 21n + 6 \rightarrow O(n^3) \text{ ma anche } O(n^4), O(n^5), \dots \\ \hookrightarrow \text{e' } O(n^c) \text{ con } c \geq 3$$

Notazioni O , Ω , Θ

Definizione – Notazione Ω

Sia $g(n)$ una funzione di costo; indichiamo con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geq 0 : f(n) \geq cg(n), \forall n \geq m$$

- Come si legge: $f(n)$ è “**Omega grande**” di $g(n)$
- Come si scrive: $f(n) = \Omega(g(n))$
- $g(n)$ è un **limite asintotico inferiore** per $f(n)$
- $f(n)$ cresce almeno quanto $g(n)$

Cresce almeno alla stessa velocità di un certo tasso di crescita
esempio

$7n^3 + 100n^2 + 21n + 6$ questa funzione è $\Omega(n^3)$ ma anche $\Omega(n^2)$ $\Omega(n)$
 \rightarrow è $\Omega(n^c)$ con $c \leq 3$

Notazioni O , Ω , Θ

Definizione – Notazione Θ

Sia $g(n)$ una funzione di costo; indichiamo con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 : c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq m$$

- Come si legge: $f(n)$ è “Theta” di $g(n)$
- Come si scrive: $f(n) = \Theta(g(n))$
- $f(n)$ cresce esattamente come $g(n)$
- $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

La notazione Θ descrive il tasso di crescita di una funzione entro un certo fattore costante dall'alto ed entro un certo fattore costante dal basso

$$7n^3 + 100n^2 + 21n + 6 \rightarrow \text{è sia } O(n^3) \text{ sia } \Omega(n^3) \Rightarrow \Theta(n^3)$$

Giustifichiamo la tecnica di scaricare i termini di ordine inferiore e ignorare il coefficiente costante del termine di ordine superiore

Obiettivo

Trovare delle costanti positive c e n_0 tali che $4n^2 + 100n + 500 \leq cn^2$ per tutti gli $n \geq n_0$

$$\rightarrow 4n^2 + 100n + 500 = O(n^2)$$

$$4n^2 + 100n + 500 \leq cn^2$$

$$4 + \frac{100}{n} + \frac{500}{n^2} \leq c$$

$$n_0 = 10$$

$$4 + 10 + 5 \leq c \rightarrow c \geq 19$$

$$n \geq n_0 \rightarrow n = 100$$

$$4 \cdot 100^2 + 100 \cdot 100 + 500 \leq 19 \cdot 100^2 \quad \text{Vero!}$$

Esempio 2

$$4n^2 + 100n + 500 = O(n)$$

$$4n^2 + 100n + 500 \leq cn$$

$$4n + 100 + \frac{500}{n} \leq c$$

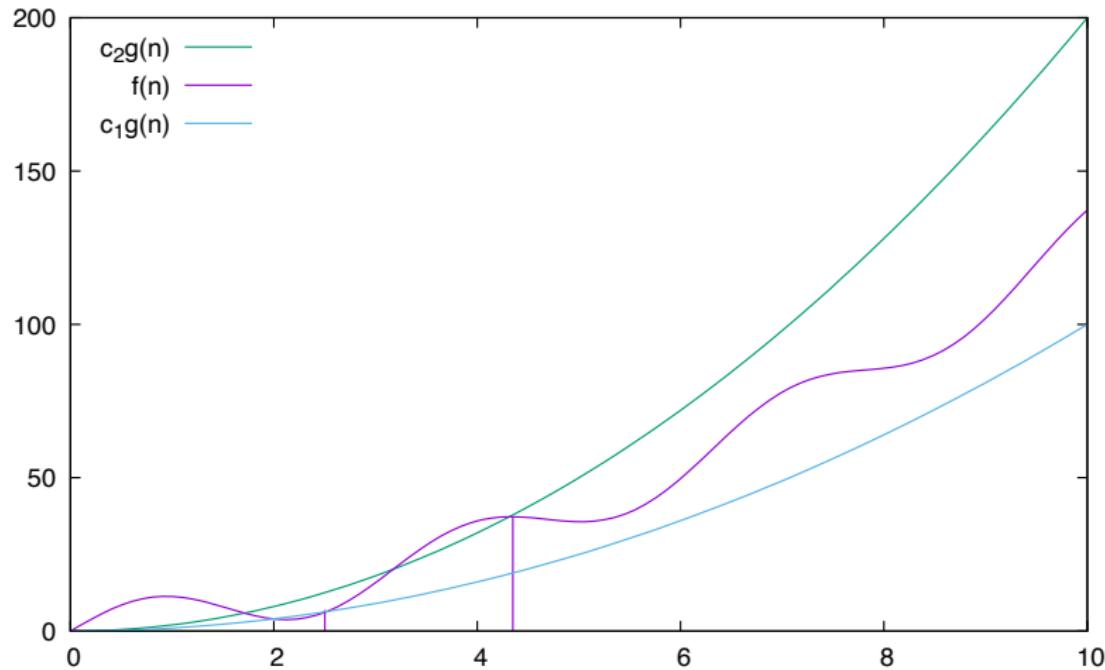
$$n_0 = 10$$

$$4 \cdot 10 + 100 + 50 \leq c \rightarrow 190 \leq c$$

$$n = 100$$

$$4 \cdot 100^2 + 100 \cdot 100 + 500 \leq 190 \cdot 100 \quad \text{Falso!}$$

Graficamente



Vero o falso?

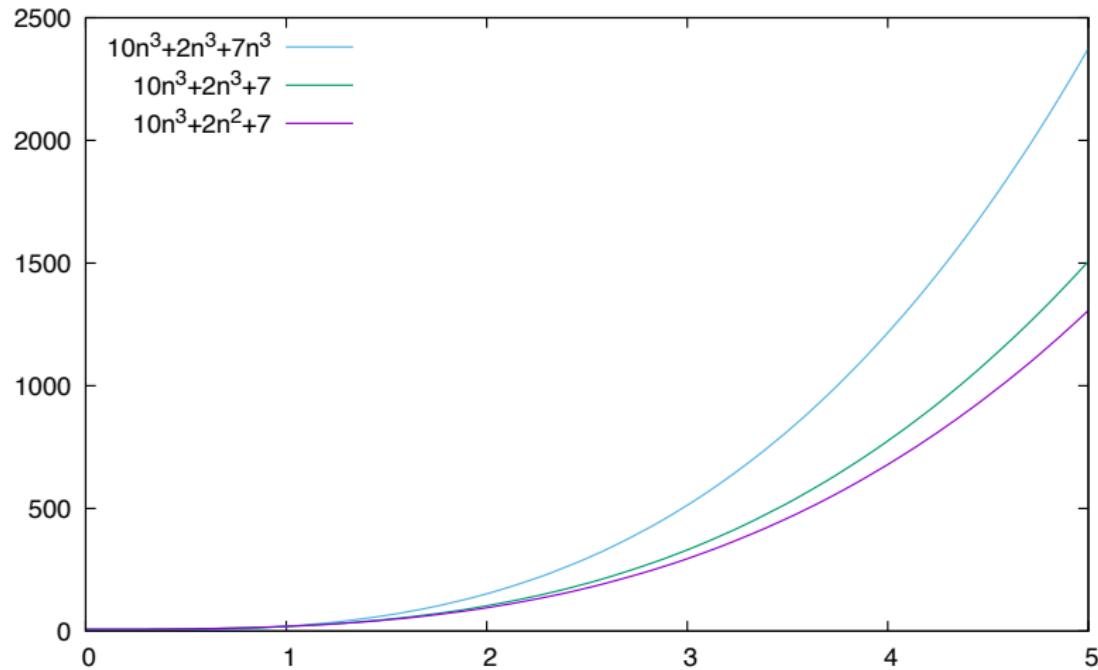
$$f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} O(n^3)$$

Dobbiamo provare che $\exists c > 0, \exists m \geq 0 : f(n) \leq cn^3, \forall n \geq m$

$$\begin{aligned} f(n) &= 10n^3 + 2n^2 + 7 \\ &\leq 10n^3 + 2n^3 + 7 && \forall n \geq 1 \\ &\leq 10n^3 + 2n^3 + 7n^3 && \forall n \geq 1 \\ &= 19n^3 \\ &\stackrel{?}{\leq} cn^3 \end{aligned}$$

che è vera per ogni $c \geq 19$ e per ogni $n \geq 1$, quindi $m = 1$.

Graficamente



Non è l'unico modo di procedere

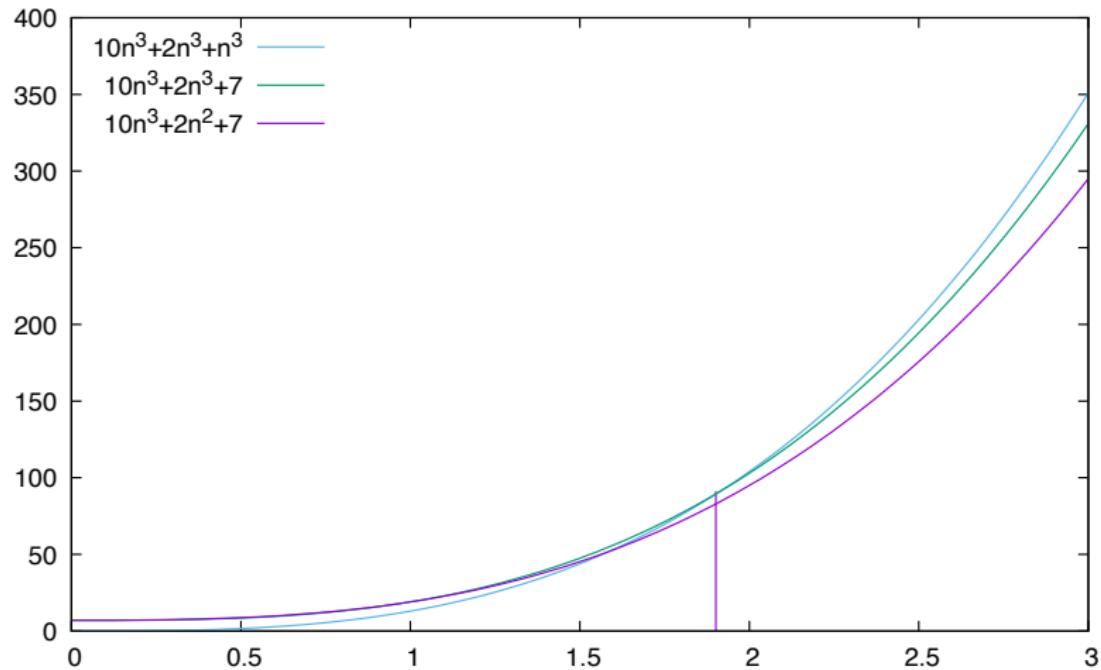
$$f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} O(n^3)$$

Dobbiamo provare che $\exists c > 0, \exists m \geq 0 : f(n) \leq cn^3, \forall n \geq m$

$$\begin{aligned} f(n) &= 10n^3 + 2n^2 + 7 \\ &\leq 10n^3 + 2n^3 + 7 && \forall n \geq 1 \\ &\leq 10n^3 + 2n^3 + n^3 && \forall n \geq \sqrt[3]{7} \\ &= 13n^3 \\ &\stackrel{?}{\leq} cn^3 \end{aligned}$$

che è vera per ogni $c \geq 13$ e per ogni $n \geq \sqrt[3]{7}$, quindi usiamo $m = 2$

Graficamente



Vero o falso?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

Limite inferiore: $\exists c_1 > 0, \exists m_1 \geq 0 : f(n) \geq c_1 n^2, \forall n \geq m_1$

Vero o falso?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

Limite inferiore: $\exists c_1 > 0, \exists m_1 \geq 0 : f(n) \geq c_1 n^2, \forall n \geq m_1$

$$\begin{aligned} f(n) &= 3n^2 + 7n \\ &\geq 3n^2 && \text{Per } n \geq 0 \\ &\stackrel{?}{\geq} c_1 n^2 \end{aligned}$$

che è vera per ogni $c_1 \leq 3$ e per ogni $n \geq 0$, quindi $m_1 = 0$

Vero o falso?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

Limite superiore: $\exists c_2 > 0, \exists m_2 \geq 0 : f(n) \leq c_2 n^2, \forall n \geq m_2$

Vero o falso?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

Limite superiore: $\exists c_2 > 0, \exists m_2 \geq 0 : f(n) \leq c_2 n^2, \forall n \geq m_2$

$$\begin{aligned} f(n) &= 3n^2 + 7n \\ &\leq 3n^2 + 7n^2 && \text{Per } n \geq 1 \\ &= 10n^2 \\ &\stackrel{?}{\leq} c_2 n^2 \end{aligned}$$

che è vera per ogni $c_2 \geq 10$ e per ogni $n \geq 1$, quindi $m_2 = 1$

Vero o falso?

$$f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)$$

Notazione Θ :

$$\exists c_1 > 0, \exists c_2 > 0, \exists m \geq 0 : c_1 n^2 \leq f(n) \leq c_2 n^2, \forall n \geq m$$

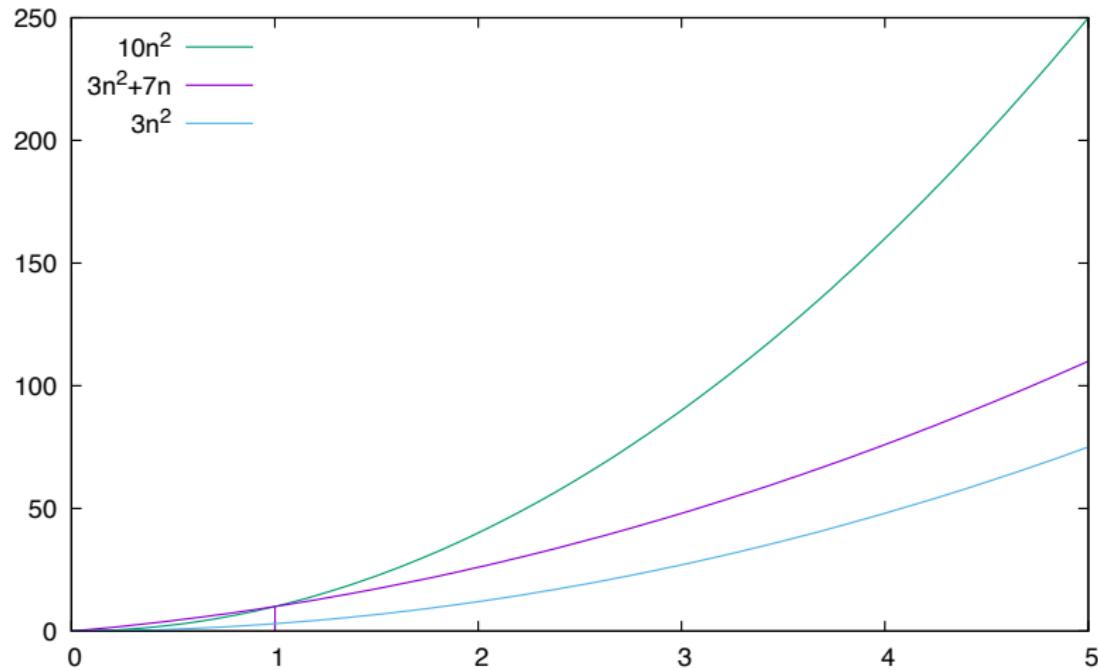
Con questi parametri:

$$c_1 = 3$$

$$c_2 = 10$$

$$m = \max\{m_1, m_2\} = \max\{0, 1\} = 1$$

Graficamente



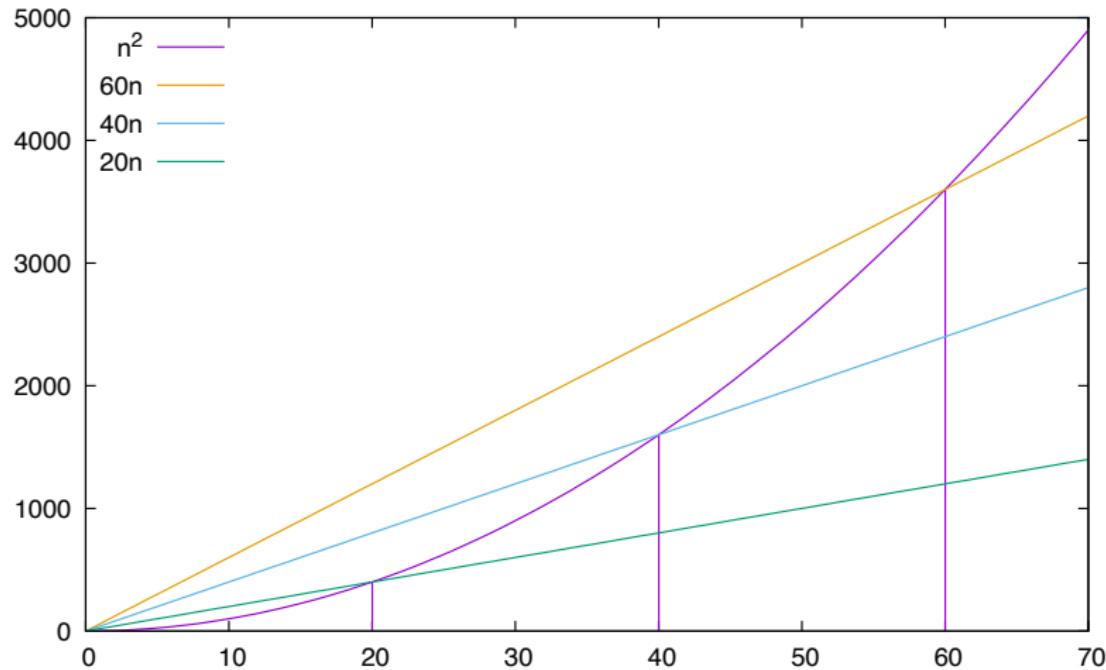
Vero o falso?

$$n^2 \stackrel{?}{=} O(n)$$

Dobbiamo dimostrare che $\exists c > 0, \exists m > 0 : n^2 \leq cn, \forall n \geq m$

- Otteniamo questo: $n^2 \leq cn \Leftrightarrow c \geq n$
- Questo significa che c cresce con il crescere di n , ovvero che non possiamo scegliere una costante c

Graficamente



Vero o falso?

$$n^2 \stackrel{?}{=} O(n^3)$$

Dobbiamo dimostrare che $\exists c > 0, \exists m > 0 : n^2 \leq cn^3, \forall n \geq m$

- Otteniamo questo: $n^2 \leq cn^3 \Leftrightarrow c \geq \frac{1}{n}$
- La funzione $1/n$ è monotona decrescente per $n > 0$.
- In altre parole, possiamo prendere un qualunque valore m (e.g., $m = 1$), e prendere un costante $c \geq 1/m$, come ad esempio $c = 1$.

Algoritmi e strutture dati

Analisi di algoritmi

Costo algoritmi vs Complessità problemi

Alberto Montresor and Davide Rossi

Università di Bologna

30 settembre 2024

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

1 Modelli di calcolo

- Definizioni
- Esempi di analisi
- Ordini di costo

2 Notazione asintotica

- Definizioni
- Esercizi

3 Costo problemi vs algoritmi

- Sommare numeri binari
- Moltiplicare numeri binari

4 Tipologia dell'input

- Selection Sort
- Insertion Sort
- Merge Sort

Introduzione

Obiettivo: riflettere su complessità di problemi/algoritmi

- In alcuni casi, si può migliorare quanto si ritiene "normale"
- In altri casi, è impossibile fare di meglio
- Qual è il rapporto fra un problema computazionale e l'algoritmo?

Back to basics!

- Somme
- Moltiplicazioni

Sommare numeri binari

Algoritmo elementare della somma – `sum()`

- richiede di esaminare tutti gli n bit
- costo totale $cn = O(n)$
($c \equiv$ costo per sommare tre bit e generare riporto)

Domanda

Esiste un metodo più efficiente?

$$\begin{array}{r}
 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & + \\
 & \swarrow & \\
 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
 & \downarrow & \\
 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
 \hline
 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
 \end{array}$$

Limite superiore alla complessità di un problema

Notazione $O(f(n))$ – Limite superiore

Un problema ha complessità $O(f(n))$ se esiste almeno un algoritmo che ha costo $O(f(n))$

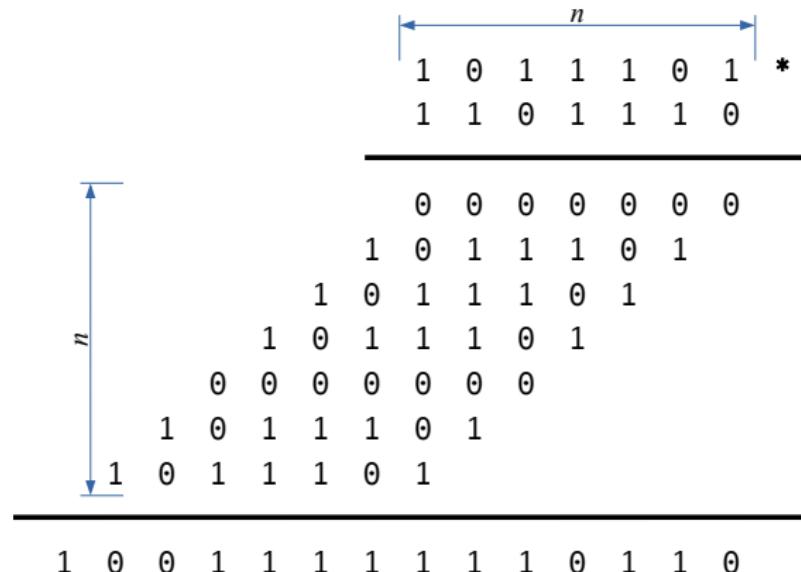
Limite superiore della somma di numeri binari

Il problema della somma di numeri binari ha complessità $O(n)$.

Moltiplicare numeri binari

Algoritmo elementare del prodotto – `prod()`

- moltiplicazione di ogni bit con ogni altro bit
- costo totale $cn^2 = O(n^2)$



Algoritmi aritmetici

Confronto della complessità computazionale

- Somma : $T_{sum}(n) = O(n)$
- Prodotto : $T_{prod}(n) = O(n^2)$

Si potrebbe concludere che...

- Il problema della moltiplicazione è inherentemente più costoso del problema dell'addizione
- Conferma la nostra esperienza

Algoritmi aritmetici

Confronto fra problemi

Per provare che il problema del prodotto è più costoso del problema della somma, dobbiamo provare che **non esiste** una soluzione in tempo sub-quadratico per il prodotto

- Abbiamo confrontato gli algoritmi, non i problemi!
- Sappiamo solo che l'algoritmo di somma delle elementari è più efficiente dell'algoritmo del prodotto delle elementari

Un po' di storia

- Nel 1960, Kolmogorov enunciò in una conferenza che la moltiplicazione ha limite inferiore $\Omega(n^2)$
- Una settimana dopo, un suo studente provò il contrario!

Moltiplicazione di Karatsuba (1962)

$$A_1 = a \times c$$

$$A_3 = b \times d$$

$$m = (a + b) \times (c + d) = ac + ad + bc + bd$$

$$A_2 = m - A_1 - A_3 = ad + bc$$



boolean [] KARATSUBA(boolean[] X, boolean[] Y, int n)

if $n == 1$ **then**

return $X[1] \cdot Y[1]$

else

spezza X in $a; b$ e Y in $c; d$

boolean[] $A_1 = \text{KARATSUBA}(a, c, n/2)$

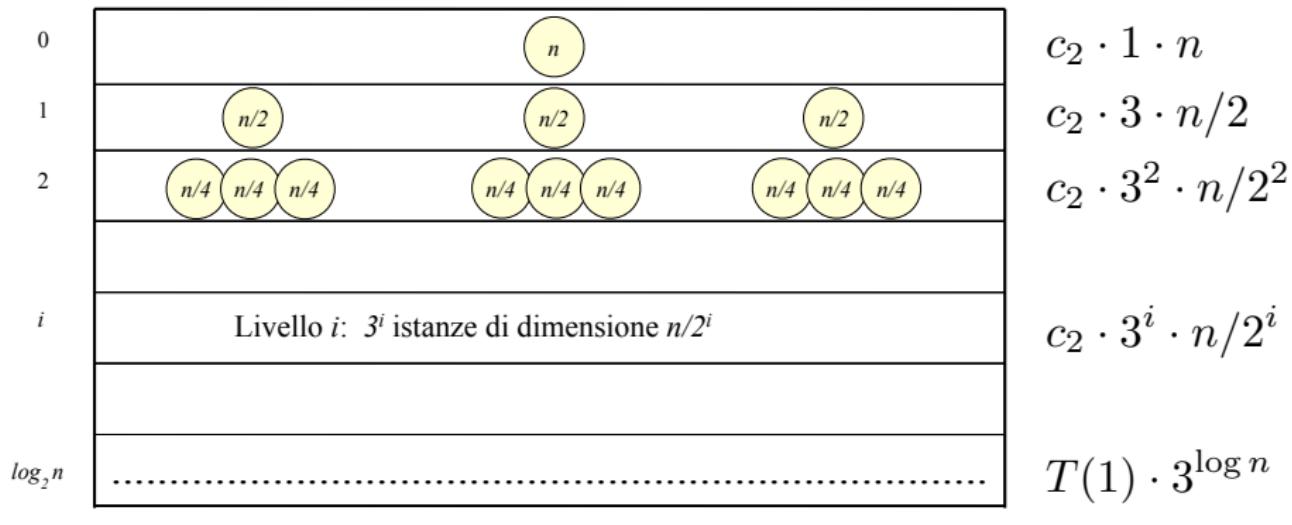
boolean[] $A_3 = \text{KARATSUBA}(b, d, n/2)$

boolean[] $m = \text{KARATSUBA}(a + b, c + d, n/2)$

boolean[] $A_2 = m - A_1 - A_3$

return $A_1 \cdot 2^n + A_2 \cdot 2^{n/2} + A_3$

Analisi della ricorsione



$$T(n) = \begin{cases} c_1 & n = 1 \\ 3T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

$$= c_1 \cdot n^{\log 3}$$

$$= c_1 \cdot n^{1.58\dots}$$

Moltiplicare numeri binari

Confronto della complessità computazionale

- Prodotto : $T_{prod}(n) = O(n^2)$ Es. $T_{prod}(10^6) = 10^{12}$
- Prodotto : $T_{kara}(n) = O(n^{1.58\dots})$ Es. $T_{kara}(10^6) = 3 \cdot 10^9$

Conclusioni

- L'algoritmo "naif" non è sempre il migliore ...
- ... può esistere spazio di miglioramento ...
- ... a meno che non sia possibile dimostrare il contrario!

Non finisce qui ...

- Toom-Cook (1963)
 - Detto anche Toom3, ha complessità $O(n^{\log 5 / \log 3}) \approx O(n^{1.465})$
 - Karatsuba \equiv Toom2
 - Moltiplicazione normale \equiv Toom1
- Schönhage–Strassen (1971)
 - Complessità $O(n \cdot \log n \cdot \log \log n)$
 - Basato su Fast Fourier Transforms
- Fürer (2007)
 - Complessità $O(n \cdot \log n \cdot K^{O(\log^* n)})$, per qualche $K > 1$
- Harvey–van der Hoeven–Lecerf (2014)
 - Complessità $O(n \cdot \log n \cdot 8^{O(\log^* n)})$
- Harvey–van der Hoeven (2019-2021)
 - Complessità $O(n \cdot \log n)$ ([\[Articolo\]](#)[\[Video\]](#))
- Limite inferiore: $\Omega(n \log n)$ (congettura)

Crescita funzioni

n	$\log^* n$	$\log \log n$
1	0	
2	1	0
4	2	1
16	3	2
2^{16}	4	4
$2^{2^{16}}$	5	16

Algoritmi vs problemi

Possiamo considerare un algoritmo come uno strumento per risolvere un problema computazionale ben definito

Costo in tempo di un algoritmo

La quantità di tempo richiesta per input di dimensione n

- $O(f(n))$: Per tutti gli input, l'algoritmo costa al più $f(n)$
- $\Omega(f(n))$: Per tutti gli input, l'algoritmo costa almeno $f(n)$
- $\Theta(f(n))$: L'algoritmo richiede $\Theta(f(n))$ per tutti gli input

Complessità in tempo di un problema computazionale

La complessità in tempo relativa a tutte le possibili soluzioni

- $O(f(n))$: Complessità del miglior algoritmo che risolve il problema
- $\Omega(f(n))$: Dimostrare che nessun algoritmo può risolvere il problema in tempo inferiore a $\Omega(f(n))$
- $\Theta(f(n))$: Algoritmo ottimo

Algoritmi e strutture dati

Analisi di algoritmi
Algoritmi di ordinamento

Alberto Montresor and Davide Rossi

Università di Bologna

30 settembre 2024

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

1 Modelli di calcolo

- Definizioni
- Esempi di analisi
- Ordini di costo

2 Notazione asintotica

- Definizioni
- Esercizi

3 Costo problemi vs algoritmi

- Sommare numeri binari
- Moltiplicare numeri binari

4 Tipologia dell'input

- Selection Sort
- Insertion Sort
- Merge Sort

Introduzione

Obiettivo: valutare gli algoritmi in base all'input

- In alcuni casi, gli algoritmi si comportano diversamente a seconda delle caratteristiche dell'input
- Conoscere in anticipo tali caratteristiche permette di scegliere il miglior algoritmo in quella situazione
- Il problema dell'ordinamento è una buona palestra dove mostrare questi concetti

Algoritmi d'ordinamento

- Selection Sort
- Insertion Sort
- Merge Sort

Tipologia di analisi

Analisi del caso pessimo

- La più importante
- Il tempo di esecuzione nel caso peggiore è un limite superiore al tempo di esecuzione per qualsiasi input
- Per alcuni algoritmi, il caso peggiore si verifica molto spesso
Es.: ricerca di dati non presenti in un database

Analisi del caso medio

- Difficile in alcuni casi: cosa si intende per "medio"?
- Distribuzione uniforme



Analisi del caso ottimo

- Può avere senso se si hanno informazioni particolari sull'input

Ordinamento

Problema dell'ordinamento

- **Input:** Una sequenza $A = a_1, a_2, \dots, a_n$ di n valori
- **Output:** Una sequenza $B = b_1, b_2, \dots, b_n$ che sia una permutazione di A e tale per cui $b_1 \leq b_2 \leq \dots \leq b_n$

Problema computazionale

Approccio "demente":

- Genero tutte le possibili permutazioni fino a quando non ne trovo una già ordinata

Ordinamento

Problema dell'ordinamento

- **Input:** Una sequenza $A = a_1, a_2, \dots, a_n$ di n valori
- **Output:** Una sequenza $B = b_1, b_2, \dots, b_n$ che sia una permutazione di A e tale per cui $b_1 \leq b_2 \leq \dots \leq b_n$

Approccio "demente":

- Genero tutte le possibili permutazioni fino a quando non ne trovo una già ordinata

Approccio "naif":

- Cerco il minimo e lo metto in posizione corretta, riducendo il problema agli $n - 1$ restanti valori.

Selection Sort

Costo computazionale complessivo $\Theta(n^2)$

$$g(n) = (n-1) \cdot (n-2) + \dots$$

SelectionSort(ITEM[] A, int n)

```
for i = 1 to n - 1 do           n-1
    int min = min(A, i, n) → n-1
    A[i] ↔ A[min] n-2
```

In questo algoritmo il costo computazionale non varia ad variare della tipologia di input. Si puo' vedere che al variare dell' input il costo computazionale non cambia.

int min(ITEM[] A, int i, int n)

% Posizione del minimo parziale

```
int min = i                      1
for j = i + 1 to n do            n-1
    if A[j] < A[min] then       n-1
        % Nuovo minimo parziale n-1
        min = j
```

Il fatto che array sia in ordine crescente o decrescente il costo computazionale rimane $\Theta(n^2)$

return min

2

Selection Sort

SelectionSort(ITEM[] A, int n)

for $i = 1$ **to** $n - 1$ **do**

 | **int** $min = \min(A, i, n)$
 | $A[i] \leftrightarrow A[min]$

int min(ITEM[] A, int i, int n)

% Posizione del minimo parziale

int $min = i$
for $j = i + 1$ **to** n **do**

 | **if** $A[j] < A[min]$ **then**
 | | % Nuovo minimo parziale
 | | $min = j$
return min

 $j = 1 \quad j = 2 \quad j = 3 \quad j = 4 \quad j = 5 \quad j = 6 \quad j = 7$

$i = 1$	7	4	2	1	8	3	5
$i = 2$	1	4	2	7	8	3	5
$i = 3$	1	2	4	7	8	3	5
$i = 4$	1	2	3	7	8	4	5
$i = 5$	1	2	3	4	8	7	5
$i = 6$	1	2	3	4	5	7	8
$i = 7$	1	2	3	4	5	7	8

Selection Sort

SelectionSort(ITEM[] A, int n)

```
for i = 1 to n - 1 do
    int min = min(A, i, n)
    A[i] ↔ A[min]
```

int min(ITEM[] A, int i, int n)

```
% Posizione del minimo parziale
int min = i
for j = i + 1 to n do
    if A[j] < A[min] then
        % Nuovo minimo
        parziale
        min = j
return min
```

Complessità nel caso medio, pessimo, ottimo?

Selection Sort

SelectionSort(ITEM[] A, int n)

for $i = 1$ **to** $n - 1$ **do**
 | **int** $min = \min(A, i, n)$
 | $A[i] \leftrightarrow A[min]$

int min(ITEM[] A, int i, int n)

% Posizione del minimo parziale
int $min = i$
for $j = i + 1$ **to** n **do**
 | **if** $A[j] < A[min]$ **then**
 | % Nuovo minimo
 | parziale
 | $min = j$
return min

Complessità nel caso medio, pessimo, ottimo?

$$\sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = n^2 - n/2 = \underbrace{\underline{\underline{O(n^2)}}}$$

Insertion Sort

- Algoritmo efficiente per ordinare piccoli insiemi di elementi
- Si basa sul principio di ordinamento di una "mano" di carte da gioco (e.g. scala quaranta)

`insertionSort(ITEM[] A, int n)`

for $i = 2$ to n do	n	$\sum_{i=2}^n t_i$
ITEM $temp = A[i]$	$n - 1$	
int $j = i$	$n - 1$	
while $j > 1$ and $A[j - 1] > temp$ do	$\sum_{i=2}^n (t_{i-1})$	
$A[j] = A[j - 1]$	$\sum_{i=2}^{n-2} (t_{i-1})$	
$j = j - 1$	$\sum_{i=2}^{n-1} (t_{i-1})$	
$A[j] = temp$	$n - 1$	

Il tempo di esecuzione di un algoritmo può dipendere da quale input di quella dimensione viene scelto. Nel caso dell'`insertion sort` il caso migliore si verifica quando l'array è già ordinato
 ↳ caso migliore sostituisci $t_i = 1$
 ↳ caso peggiore (array in ordine decrescente) $t_i = i$

Risoluzione scacchierata

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

Insertion Sort

	1	2	3	4	5	6	7	<i>temp</i>
	7	4	2	1	8	3	5	
$i = 2, j = 2$	7	7	2	1	8	3	5	4
$i = 2, j = 1$	4	7	2	1	8	3	5	4
$i = 3, j = 3$	4	7	7	1	8	3	5	2
$i = 3, j = 2$	4	4	7	1	8	3	5	2
$i = 3, j = 1$	2	4	7	1	8	3	5	2

Insertion Sort

	1	2	3	4	5	6	7	<i>temp</i>
$i = 4, j = 4$	2	4	7	7	8	3	5	1
$i = 4, j = 3$	2	4	4	7	8	3	5	1
$i = 4, j = 2$	2	2	4	7	8	3	5	1
$i = 4, j = 1$	1	2	4	7	8	3	5	1
$i = 5, j = 5$	1	2	4	7	8	3	5	8
$i = 6, j = 6$	1	2	4	7	8	8	5	3

Insertion Sort

	1	2	3	4	5	6	7	<i>temp</i>
$i = 6, j = 5$	1	2	4	7	7	8	5	3
$i = 6, j = 4$	1	2	4	4	7	8	5	3
$i = 6, j = 3$	1	2	3	4	7	8	5	3
$i = 7, j = 7$	1	2	3	4	7	8	8	5
$i = 7, j = 6$	1	2	3	4	7	7	8	5
$i = 7, j = 5$	1	2	3	4	5	7	8	5

Correttezza e complessità

In questo algoritmo

- Il costo di esecuzione non dipende solo dalla dimensione...
- ma anche dalla distribuzione dei dati in ingresso

Domande

- Dimostrare che l'algoritmo è corretto
- Qual è il costo nel caso il vettore sia già ordinato?
- Qual è il costo nel caso il vettore sia ordinato in ordine inverso?
- Cosa succede "in media"? (informalmente)

Merge Sort

Divide et impera

Merge Sort è basato sulla tecnica **divide-et-impera** vista in precedenza

- **Divide**: Spezza virtualmente il vettore di n elementi in due sottovettori di $n/2$ elementi
- **Impera**: Chiama Merge Sort ricorsivamente sui due sottovettori
- **Combina**: Unisci (**merge**) le due sequenze ordinate

Idea

Si sfrutta il fatto che i due sottovettori sono già ordinati per ordinare più velocemente

Progettare Algoritmi

Per l'insertion sort abbiamo usato un approccio incrementale: ogni elemento $A[i]$ viene inserito al posto giusto nel sotto array $A[1:i]$, avendo già ordinato il sottovettore $A[1:i-1]$.

Il metodo divide et impera

Suddividono il problema in vari sottoproblemi, che sono simili al problema originale, ma di dimensioni più piccole, risolvono i sottoproblemi in modo ricorsivo, e poi, combineranno le soluzioni per costruire una soluzione del problema originale.

se Problema sufficientemente piccolo \rightarrow caso base (non usa la ricorsione)

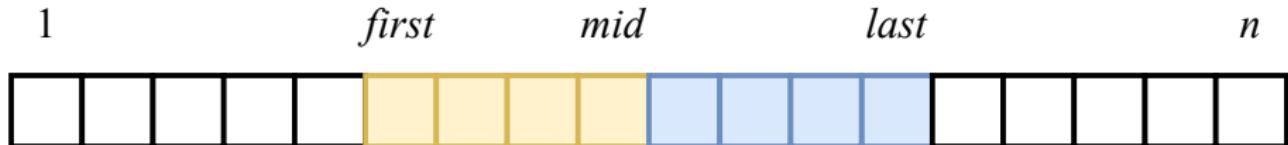
Altrimenti (caso ricorsivo) si eseguono i 3 passi caratteristici

Divide il problema viene diviso in un certo numero di sottoproblemi, che sono istanze più piccole dello stesso problema.

Impera i sottoproblemi vengono risolti in modo ricorsivo

Combina le soluzioni dei sottoproblemi vengono combinate per formare la soluzione del problema originale.

Merge



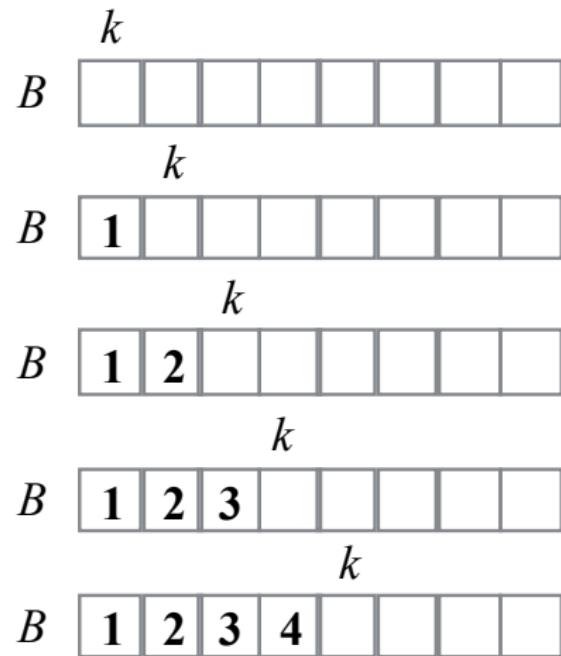
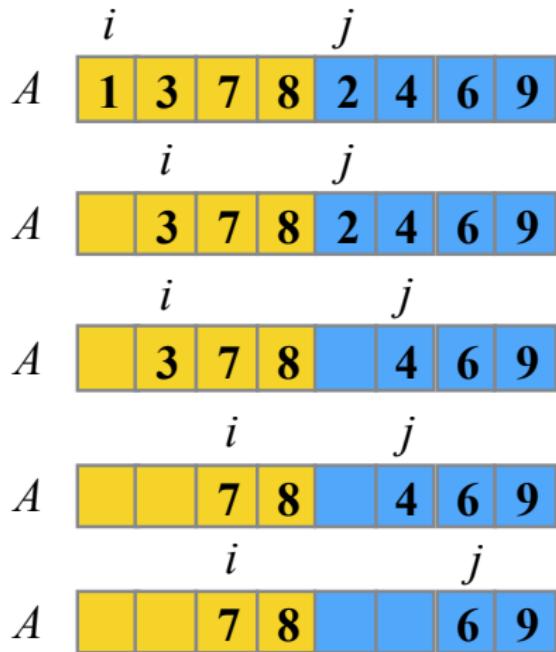
Input:

- A è un vettore di n interi
- $first, last, mid$ sono tali che $1 \leq first \leq mid < last \leq n$
- I sottovettori $A[first \dots mid]$ e $A[mid + 1 \dots last]$ sono già ordinati

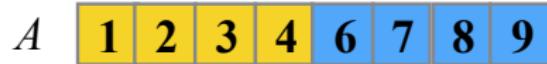
Output:

- I due sottovettori sono fusi in un unico sottovettore ordinato $A[first \dots last]$ tramite un vettore di appoggio B

Funzionamento Merge()



Funzionamento Merge()



Merge()

Merge(ITEM A[], int first, int last, int mid)

int i, j, k, h

1

$j = last$

$i = first$

1

for $h = mid$ downto i do

$j = mid + 1$

1

$A[j] = A[h]$

$k = first$

1

$j = j - 1$

while $i \leq mid$ and $j \leq last$ do

if $A[i] \leq A[j]$ then

$n+n-1$

for $j = first$ to $k - 1$ do

$B[k] = A[i]$

$A[j] = B[j]$

$i = i + 1$

Aggiunge la parte finale

else

$n+n-1$

$B[k] = A[j]$

$A[j] = B[j]$

$j = j + 1$

Aggiunge la parte iniziale ordinata

$k = k + 1$

Potrebbe riportare
una coda non ordinata
non inserita in B

vuol dire che j ha
finito
si genera B
dai sì e no
Generato
ad ogni voga
può arrivare
al massimo a
mid

Costo computazionale

Domanda

Qual è il costo computazionale di Merge()?

Costo computazionale

Domanda

Qual è il costo computazionale di Merge()? = $O(n)$

MergeSort

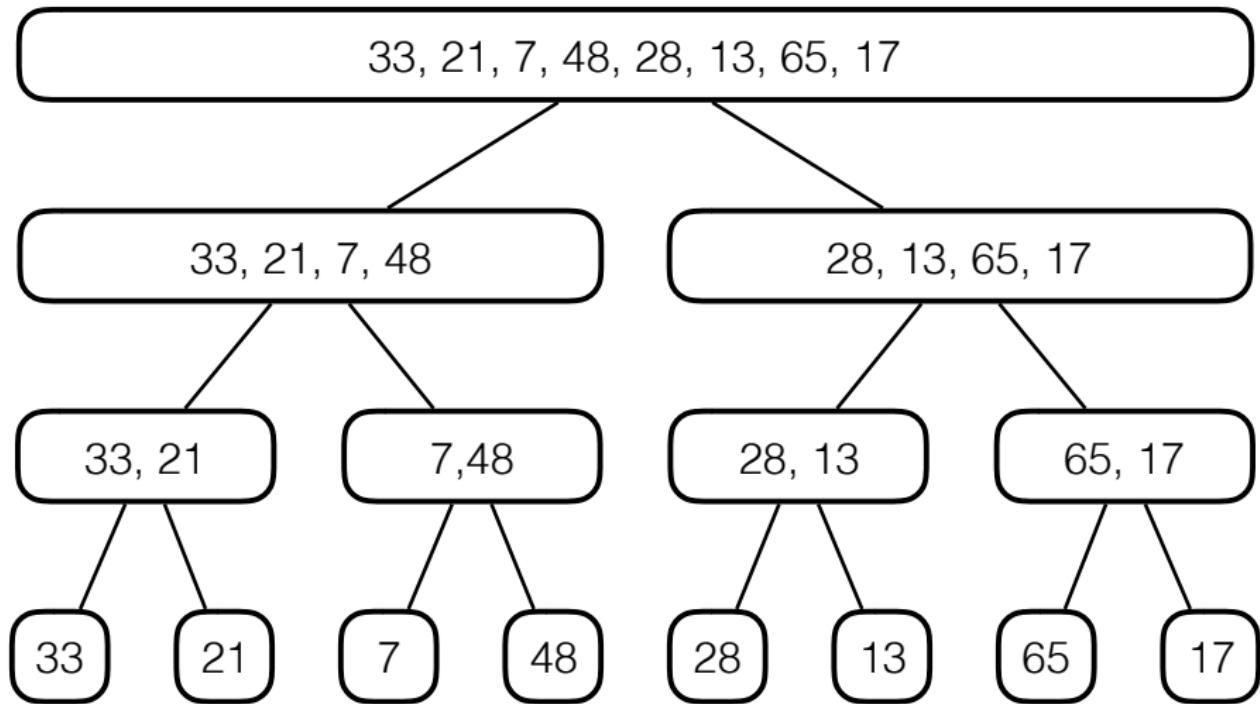
Programma completo:

- Chiama ricorsivamente se stesso e usa Merge() per unire i risultati
- Caso base: sequenze di lunghezza ≤ 1 sono già ordinate

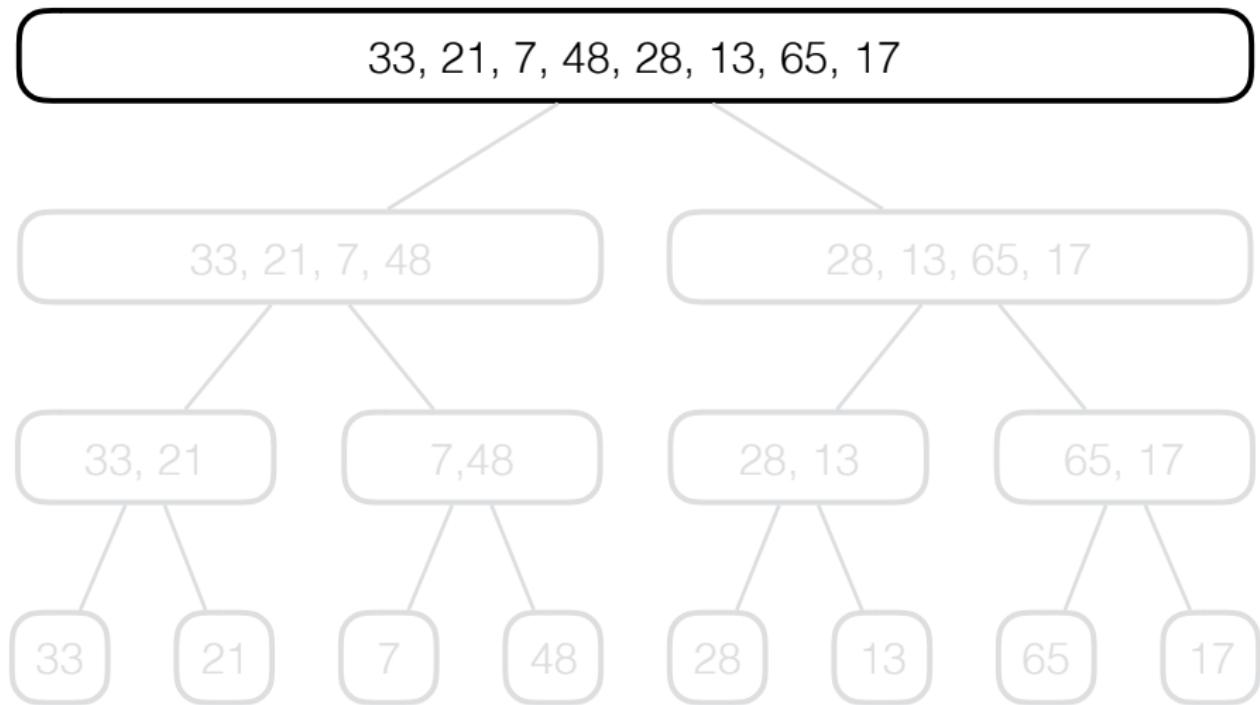
MergeSort(ITEM $A[]$, int $first$, int $last$)

```
if  $first < last$  then                                // un solo elemento
    int  $mid = \lfloor (first + last) / 2 \rfloor$            // Punto medio di  $A[first : last]$ 
    MergeSort( $A, first, mid$ )                          // Ordina ricorsivamente  $A[first : mid]$ 
    MergeSort( $A, mid + 1, last$ )                      // Ordina ricorsivamente  $A[mid + 1 : last]$ 
    Merge( $A, first, last, mid$ )                        // Fonde  $A[first : mid]$  con  $A[mid + 1 : last]$  in  $A[first : last]$ 
```

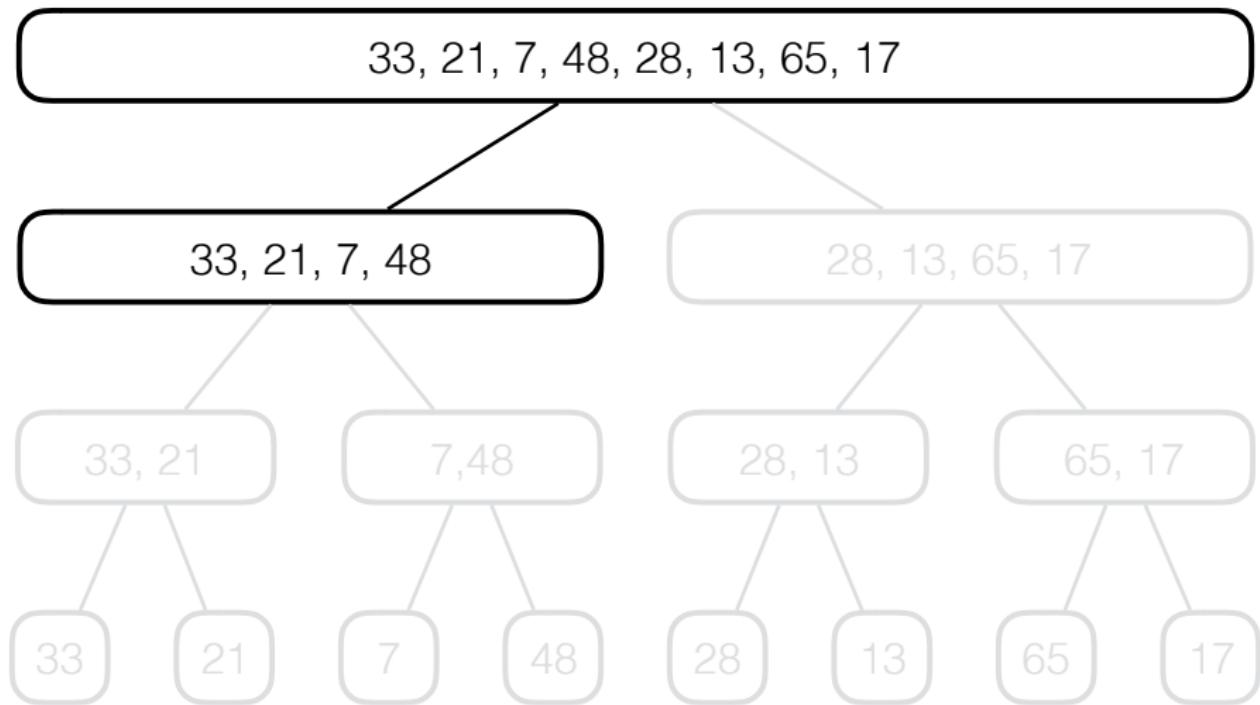
MergeSort(): Esecuzione



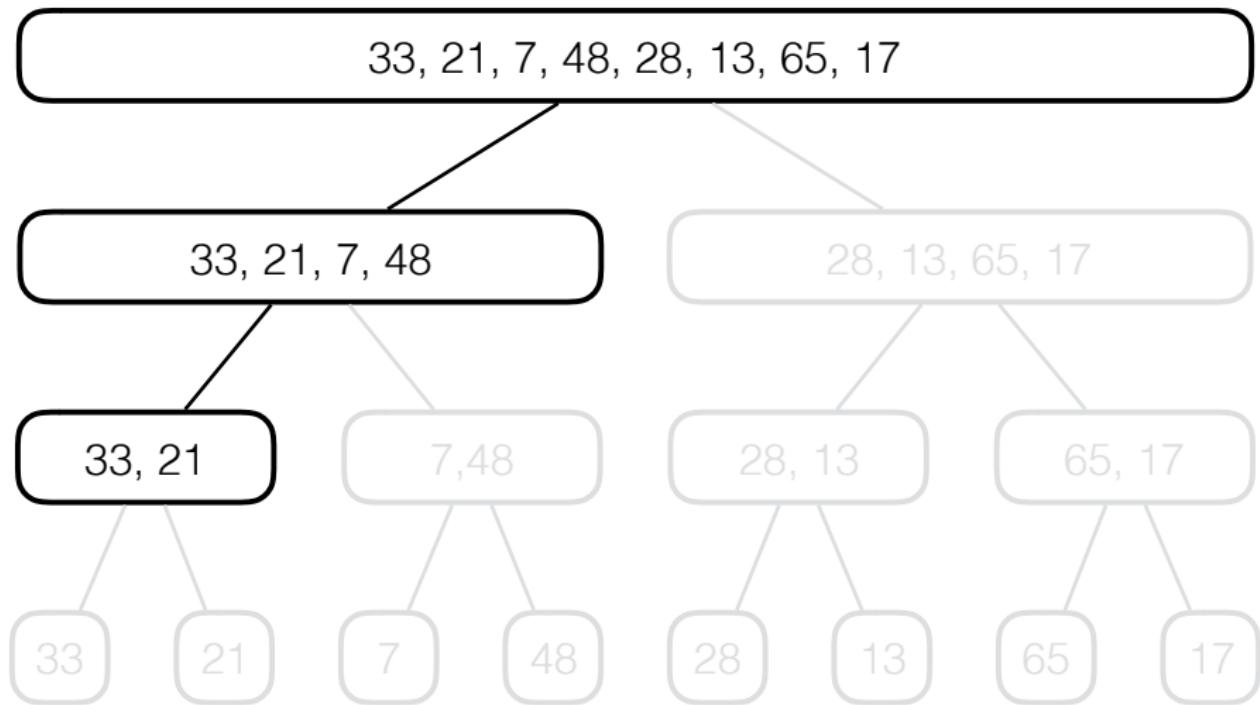
MergeSort(): Esecuzione



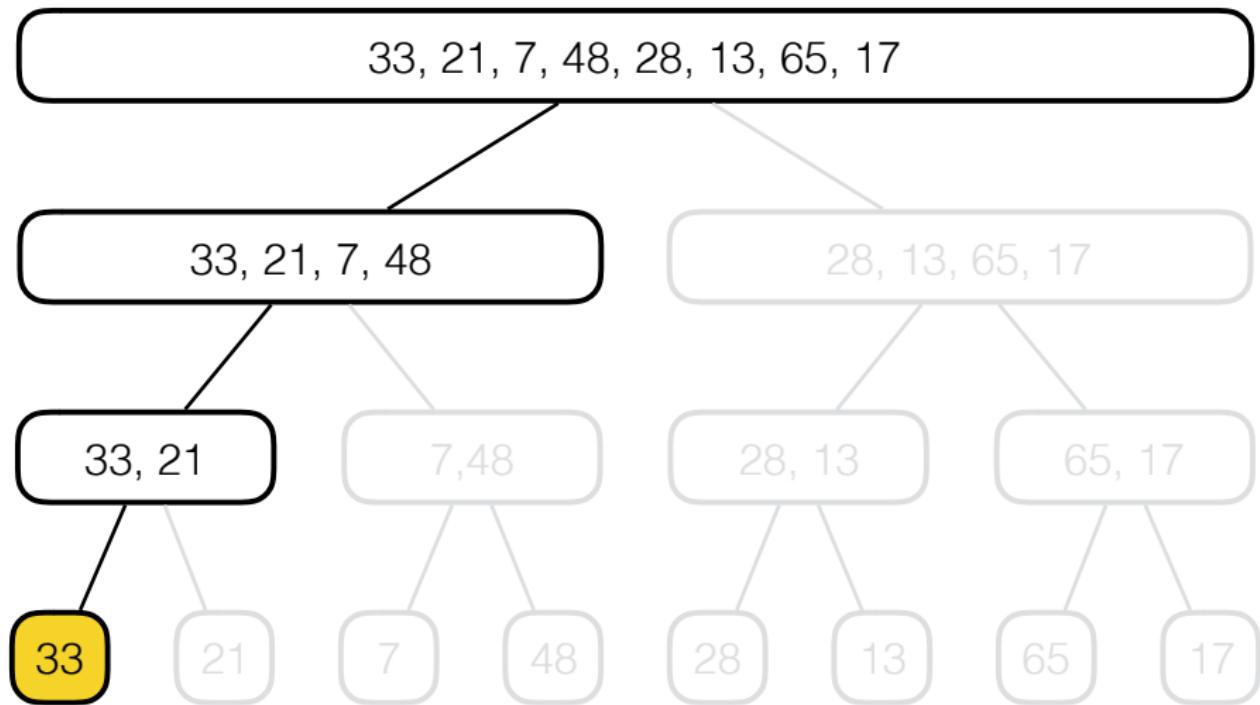
MergeSort(): Esecuzione



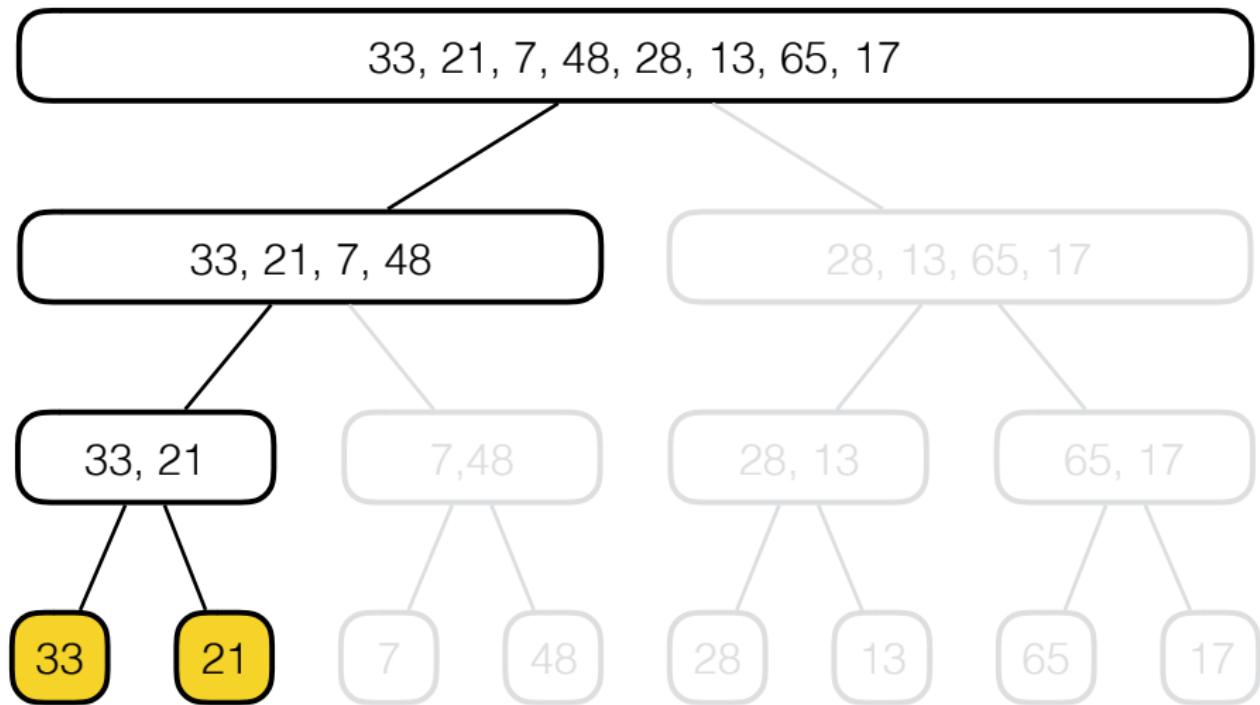
MergeSort(): Esecuzione



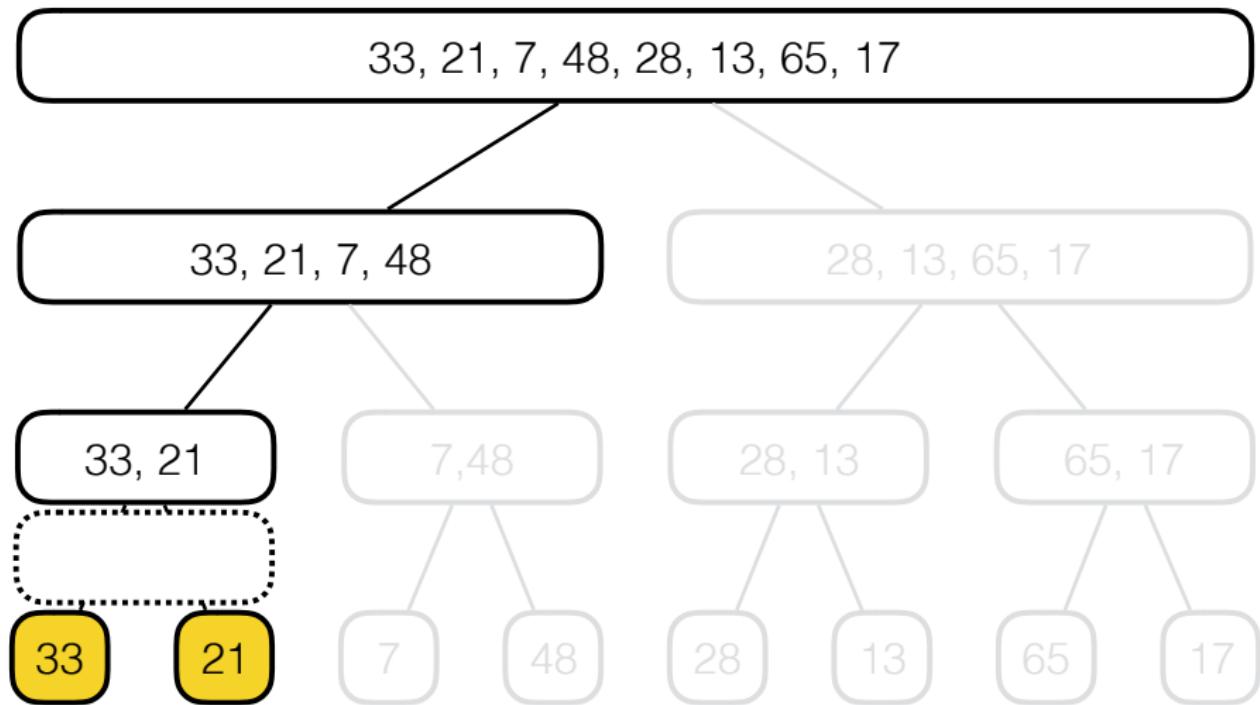
MergeSort(): Esecuzione



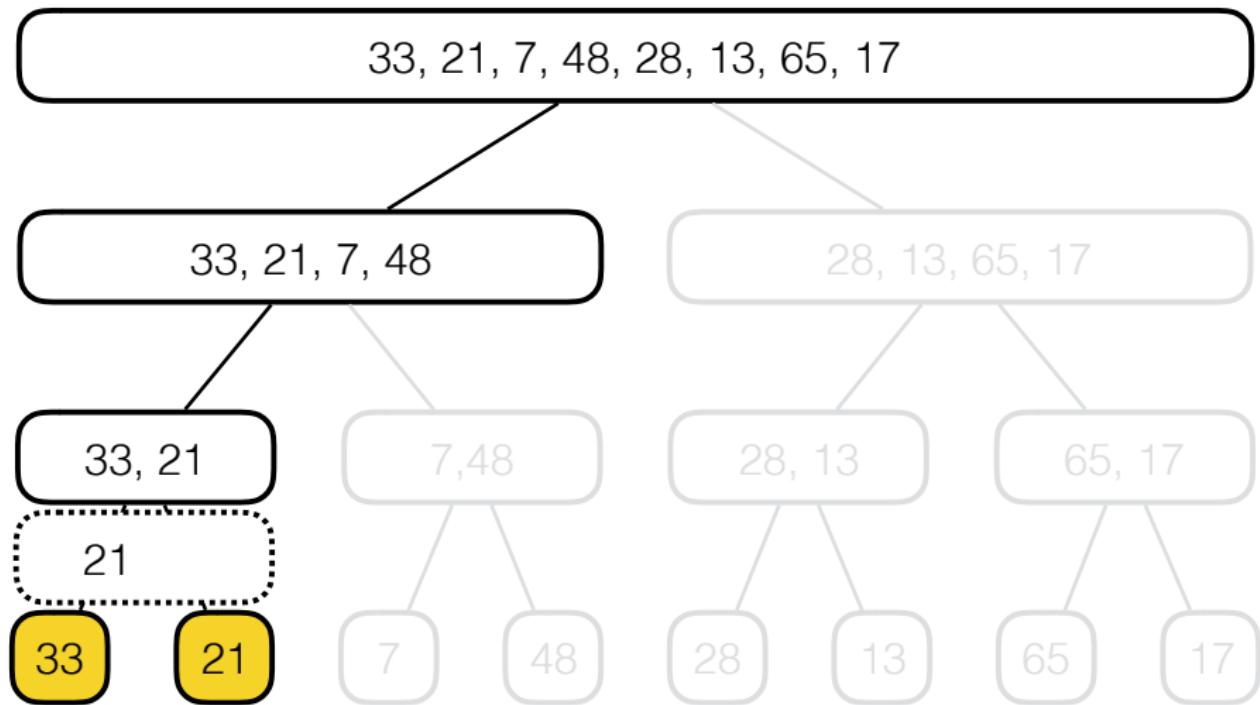
MergeSort(): Esecuzione



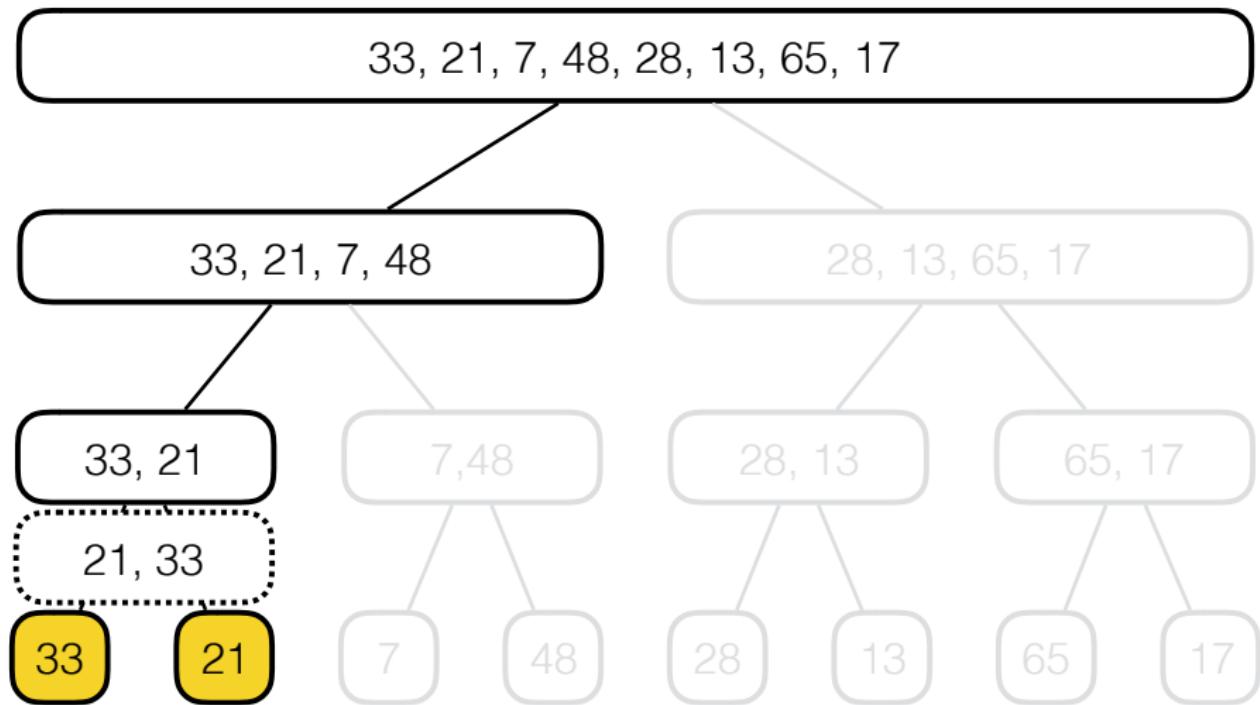
MergeSort(): Esecuzione



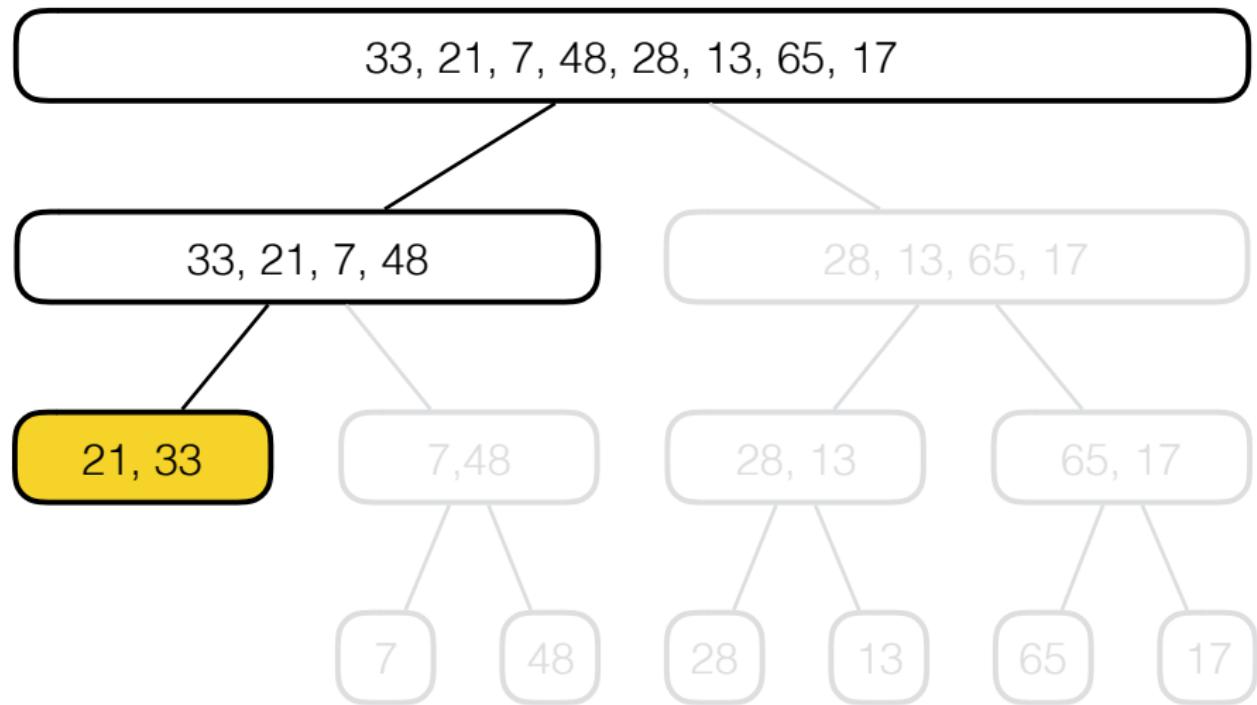
MergeSort(): Esecuzione



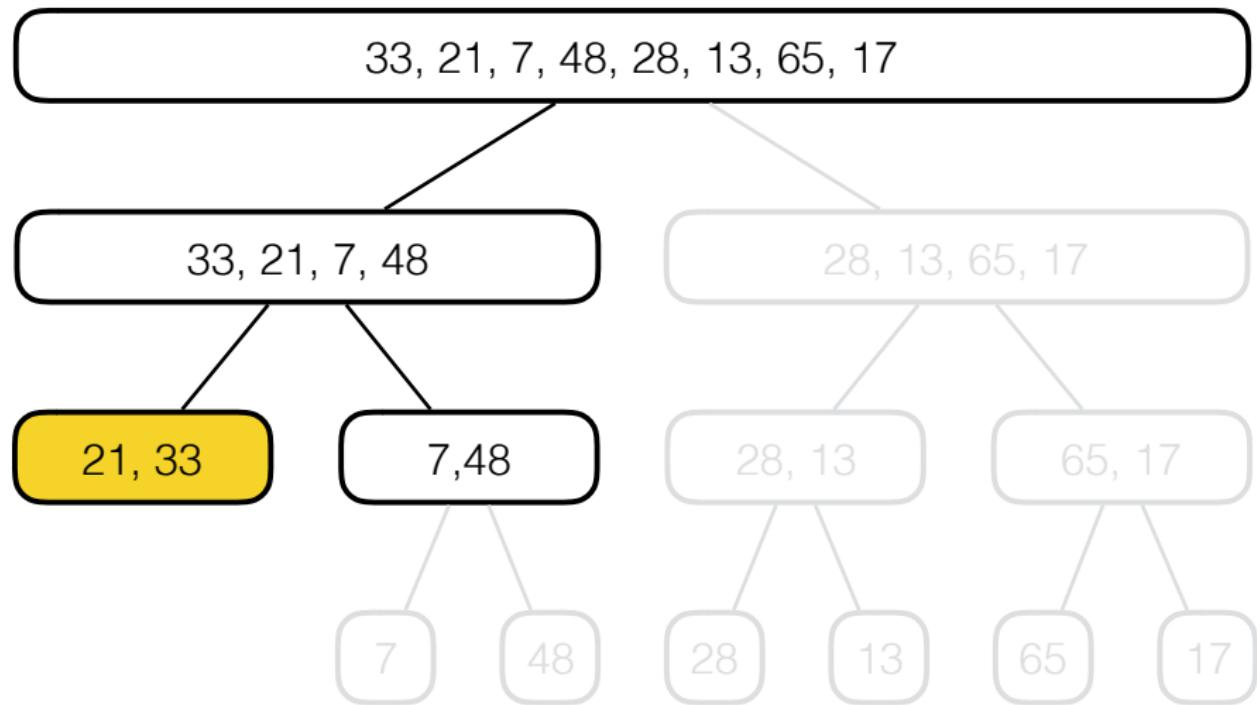
MergeSort(): Esecuzione



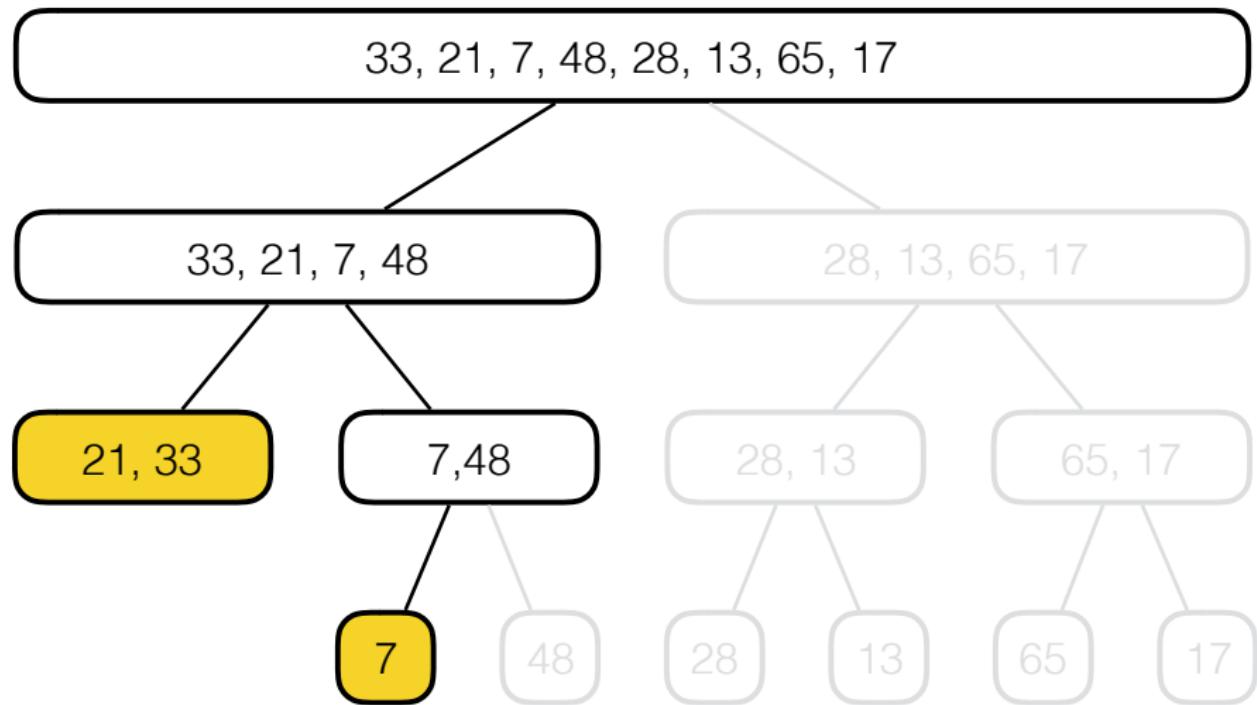
MergeSort(): Esecuzione



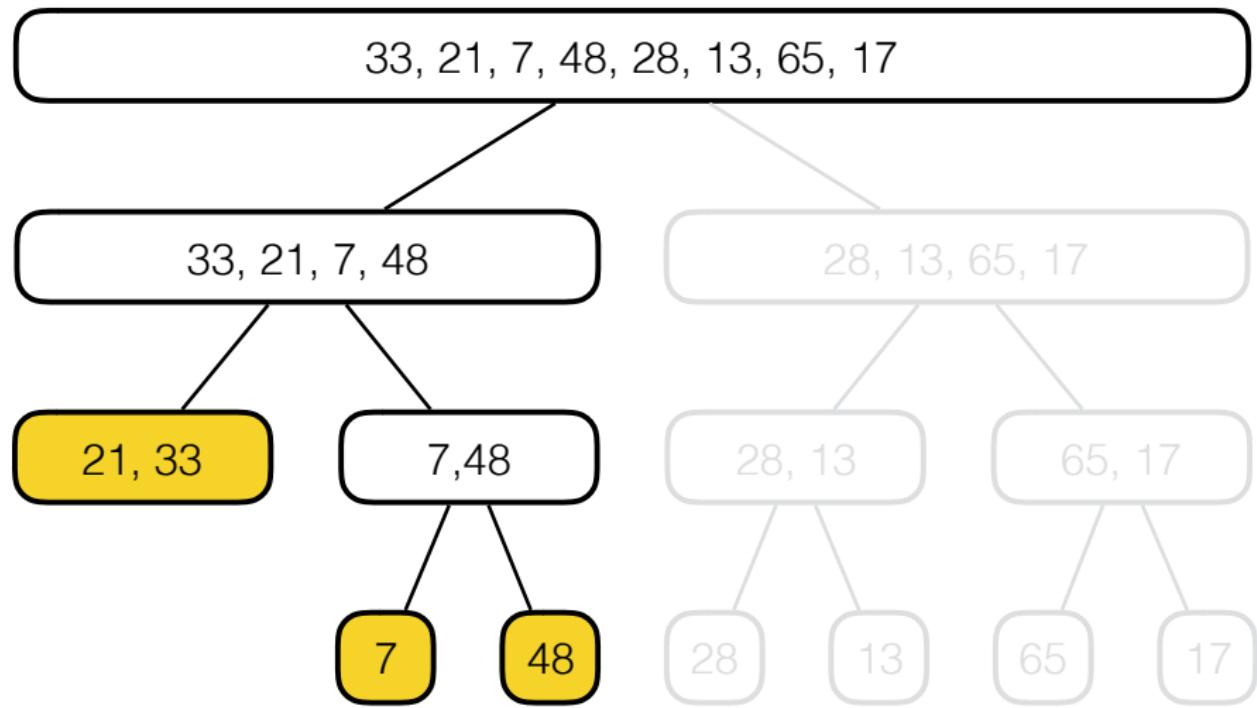
MergeSort(): Esecuzione



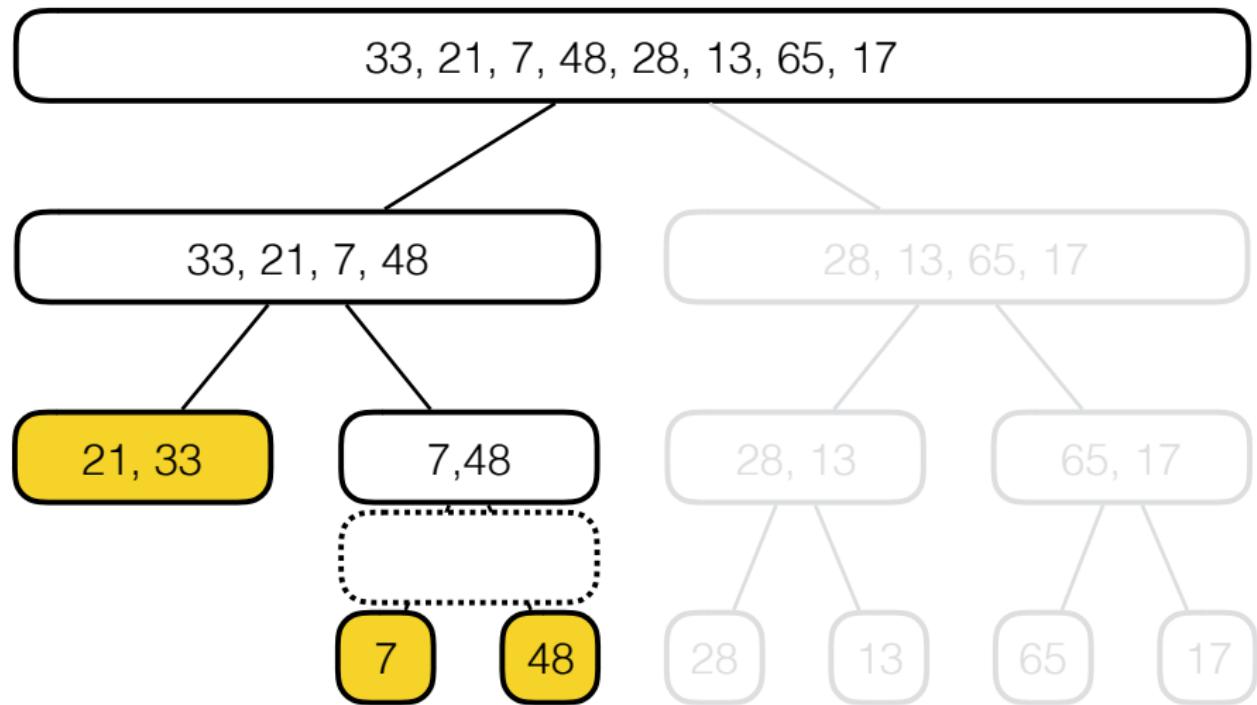
MergeSort(): Esecuzione



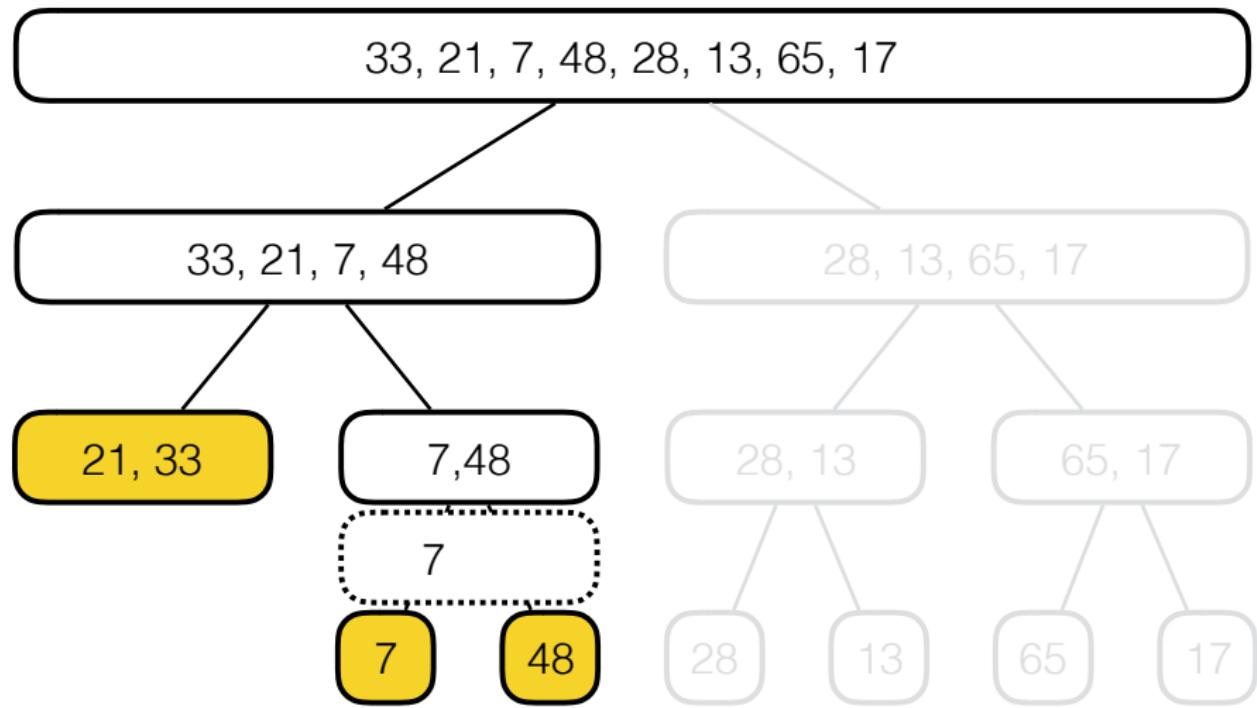
MergeSort(): Esecuzione



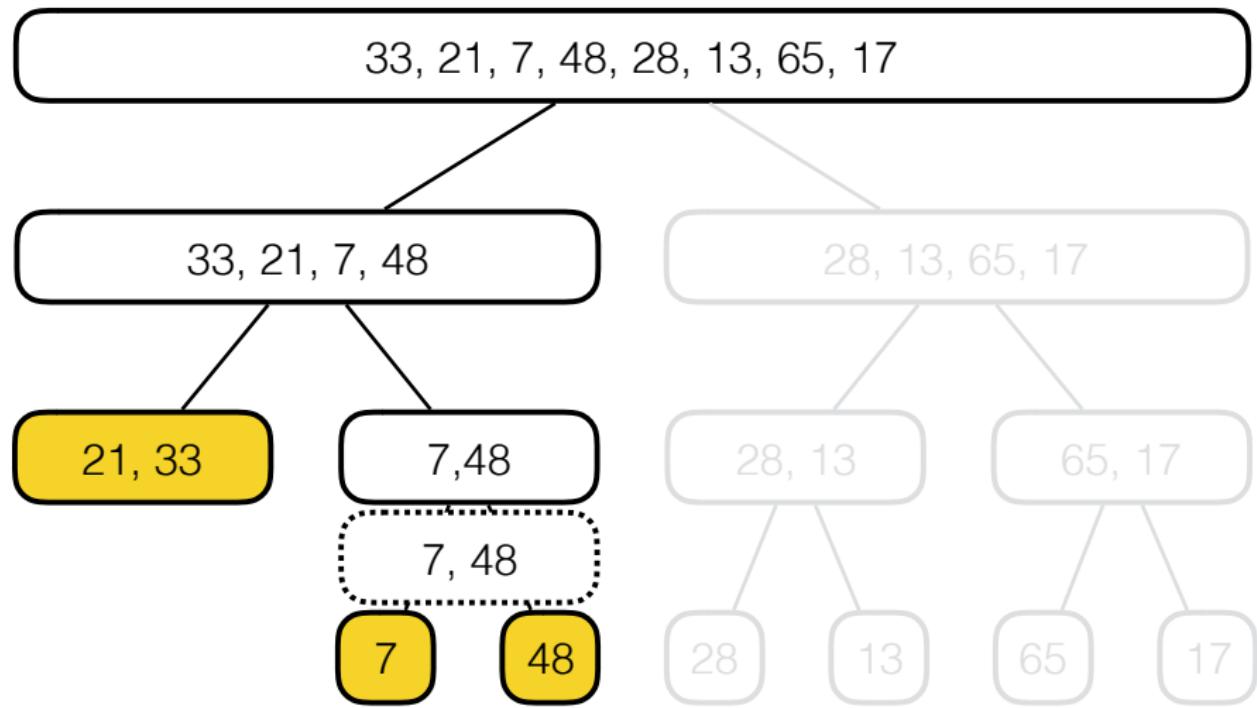
MergeSort(): Esecuzione



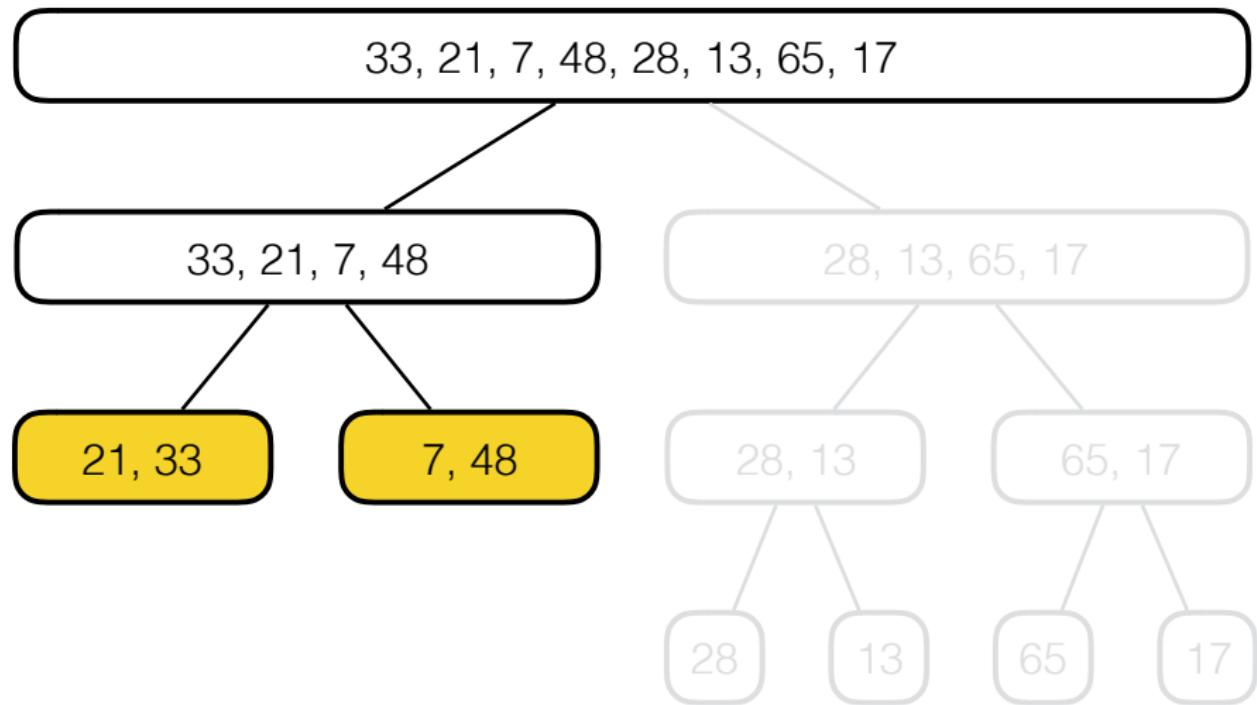
MergeSort(): Esecuzione



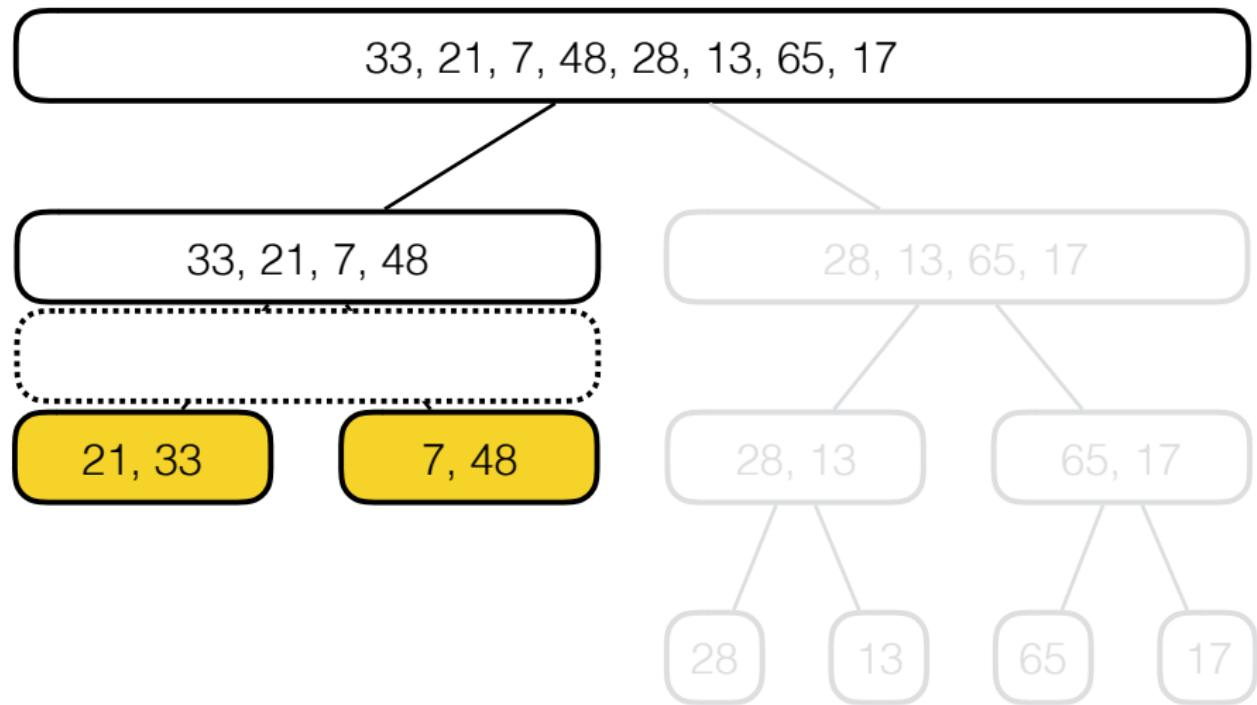
MergeSort(): Esecuzione



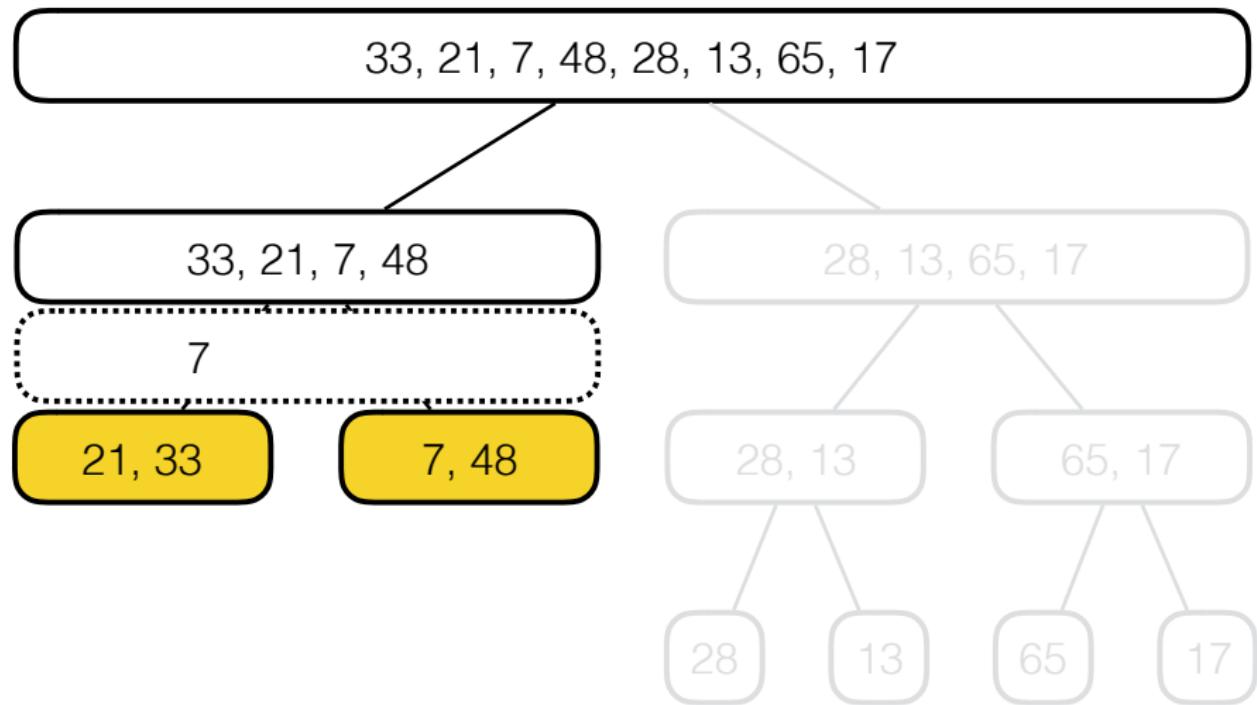
MergeSort(): Esecuzione



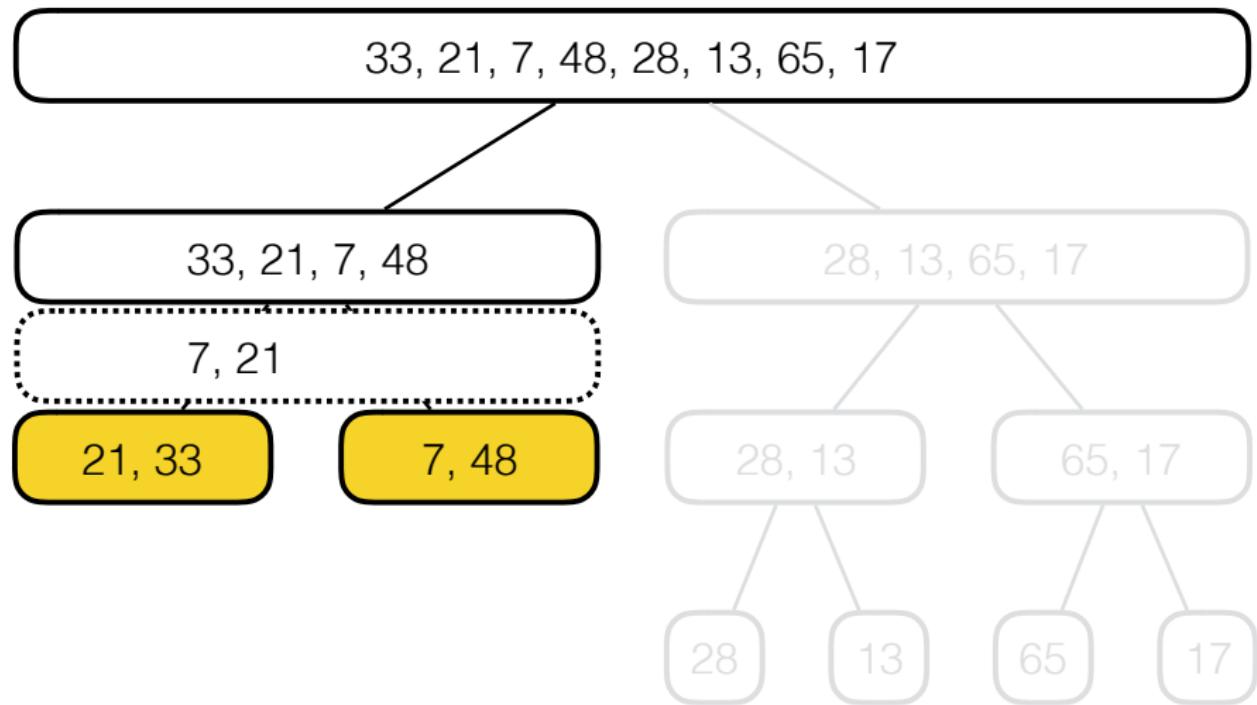
MergeSort(): Esecuzione



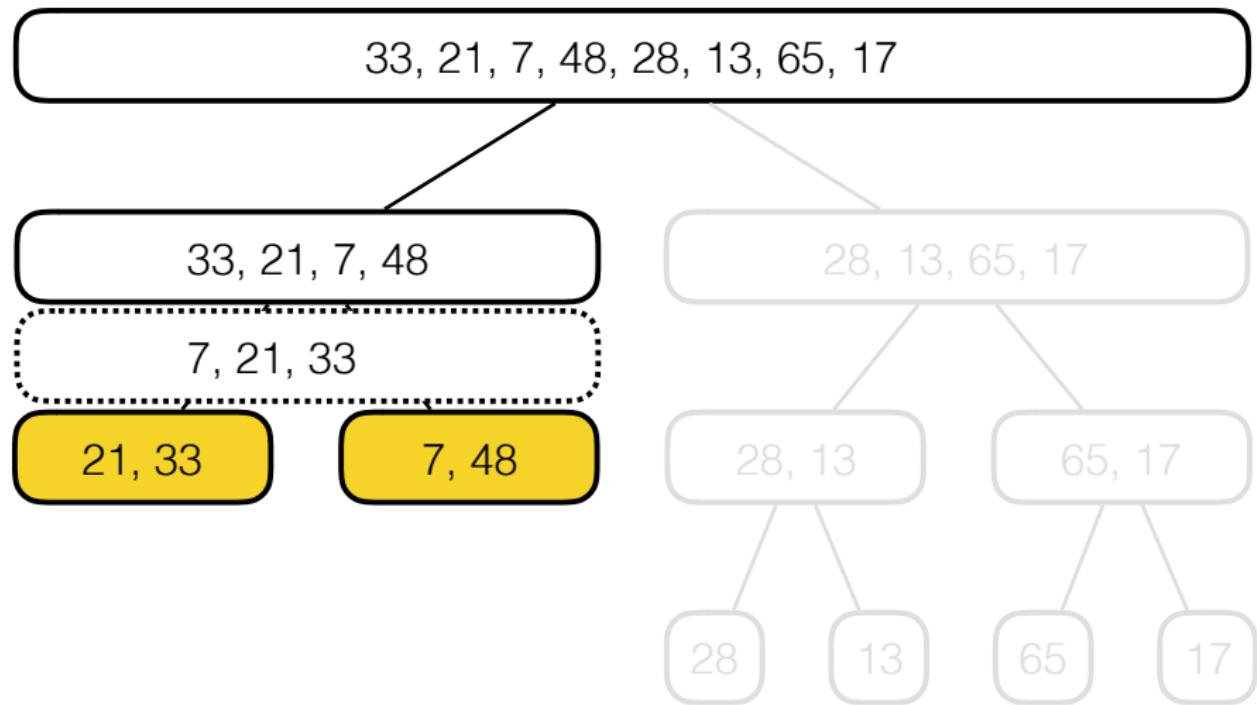
MergeSort(): Esecuzione



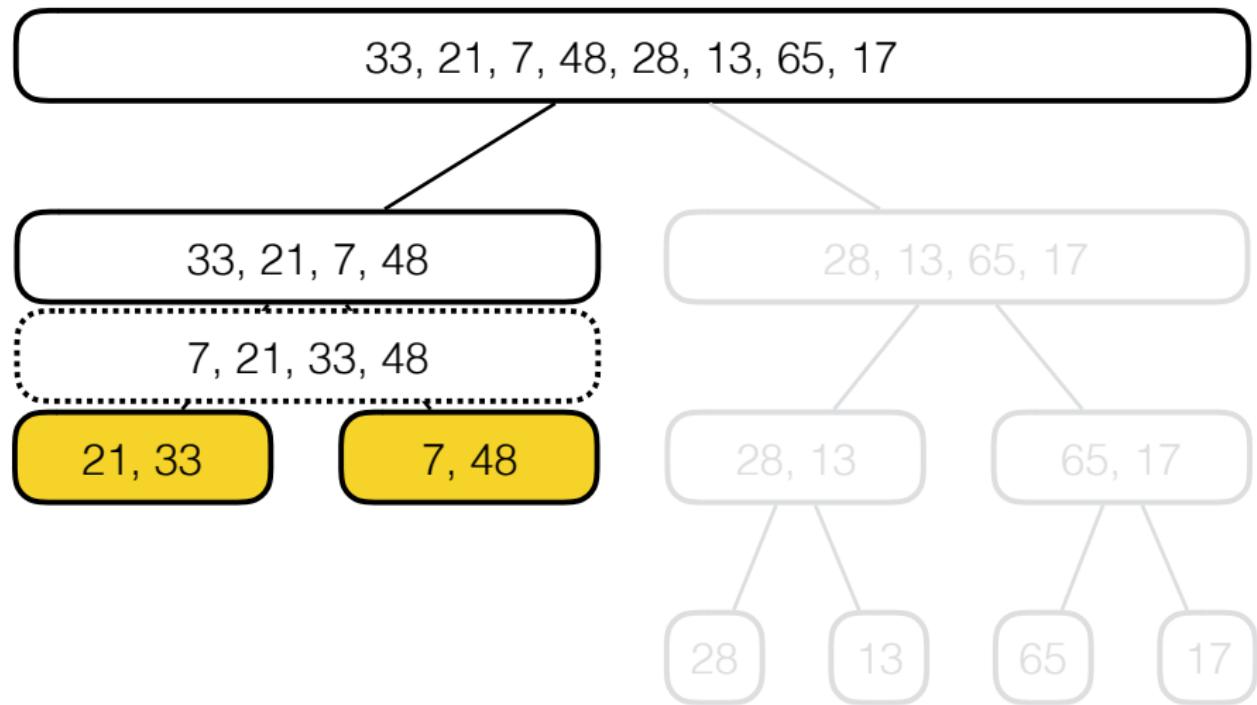
MergeSort(): Esecuzione



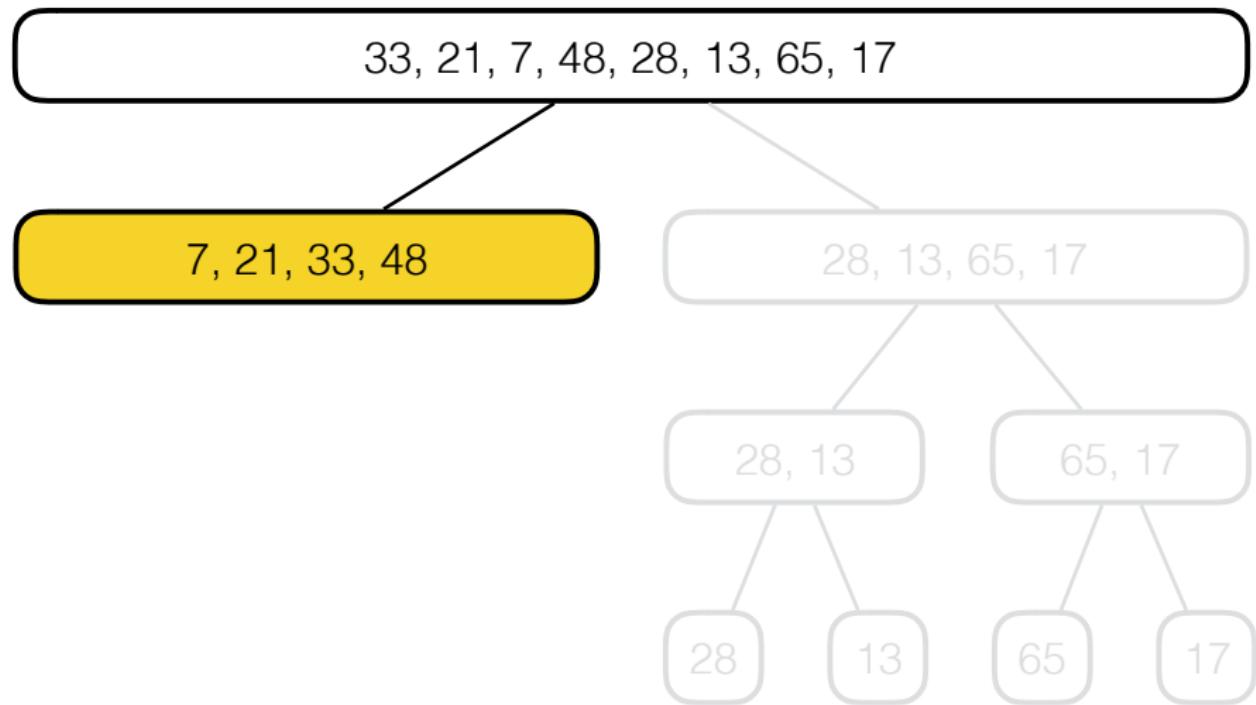
MergeSort(): Esecuzione



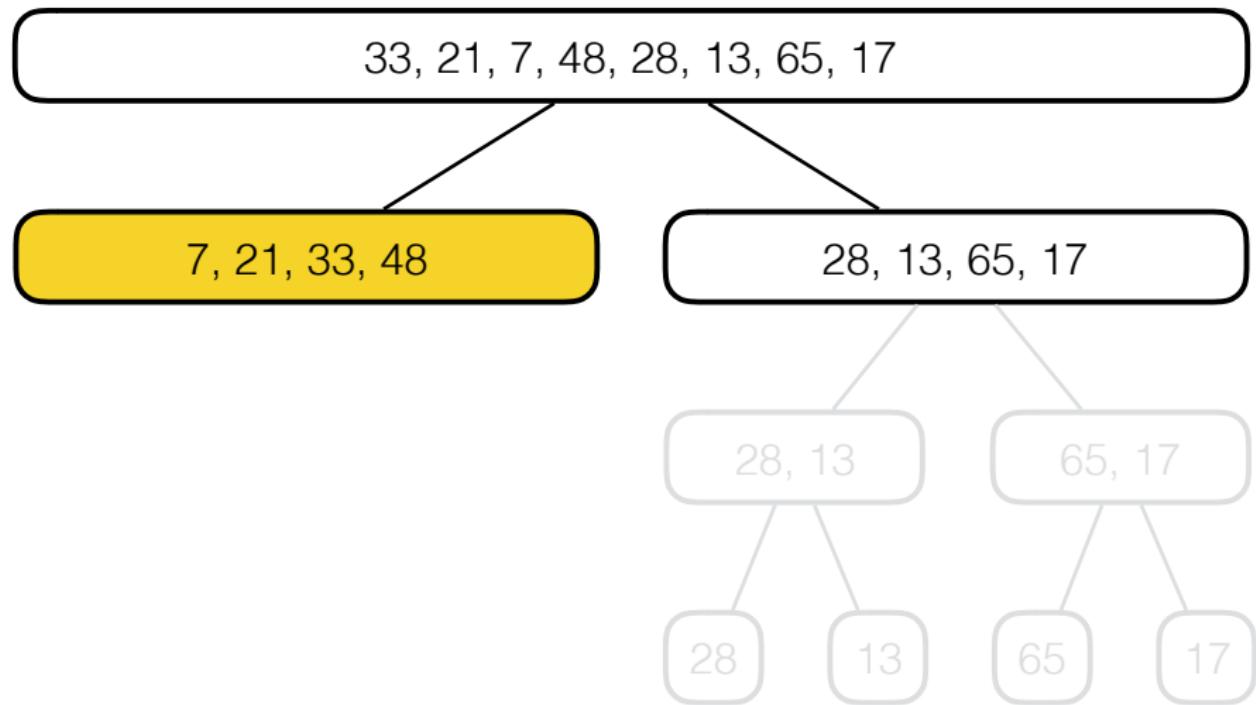
MergeSort(): Esecuzione



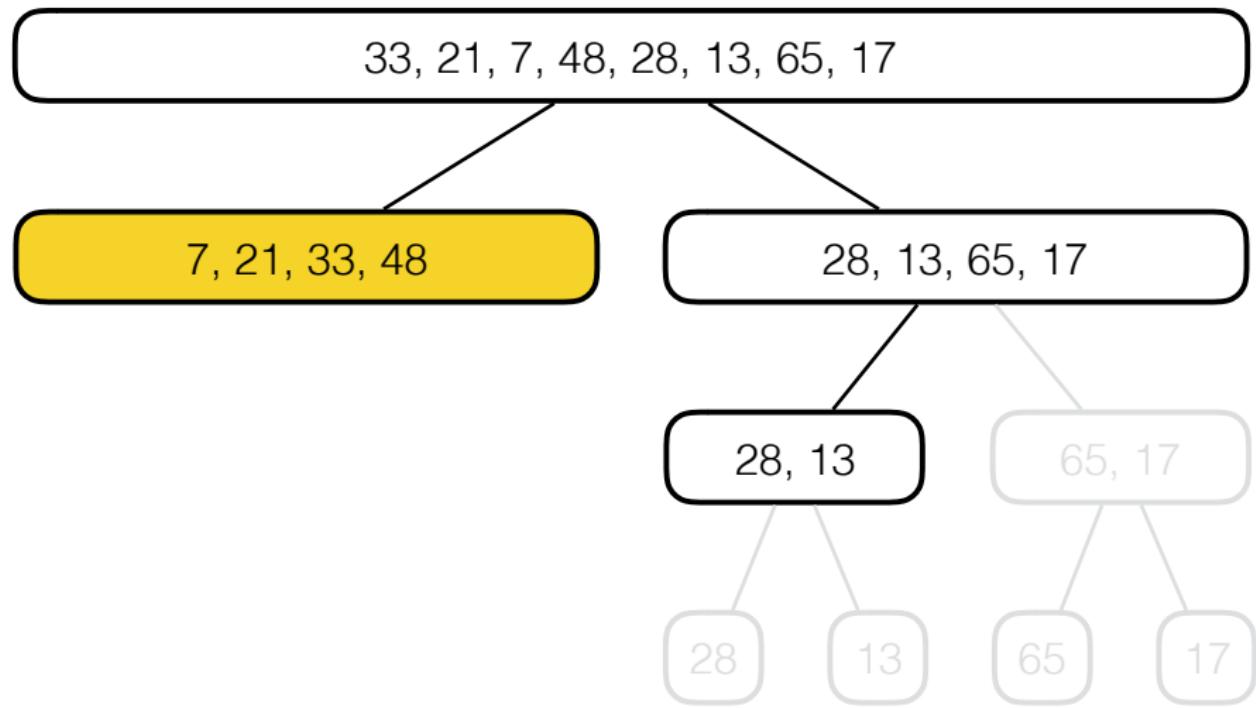
MergeSort(): Esecuzione



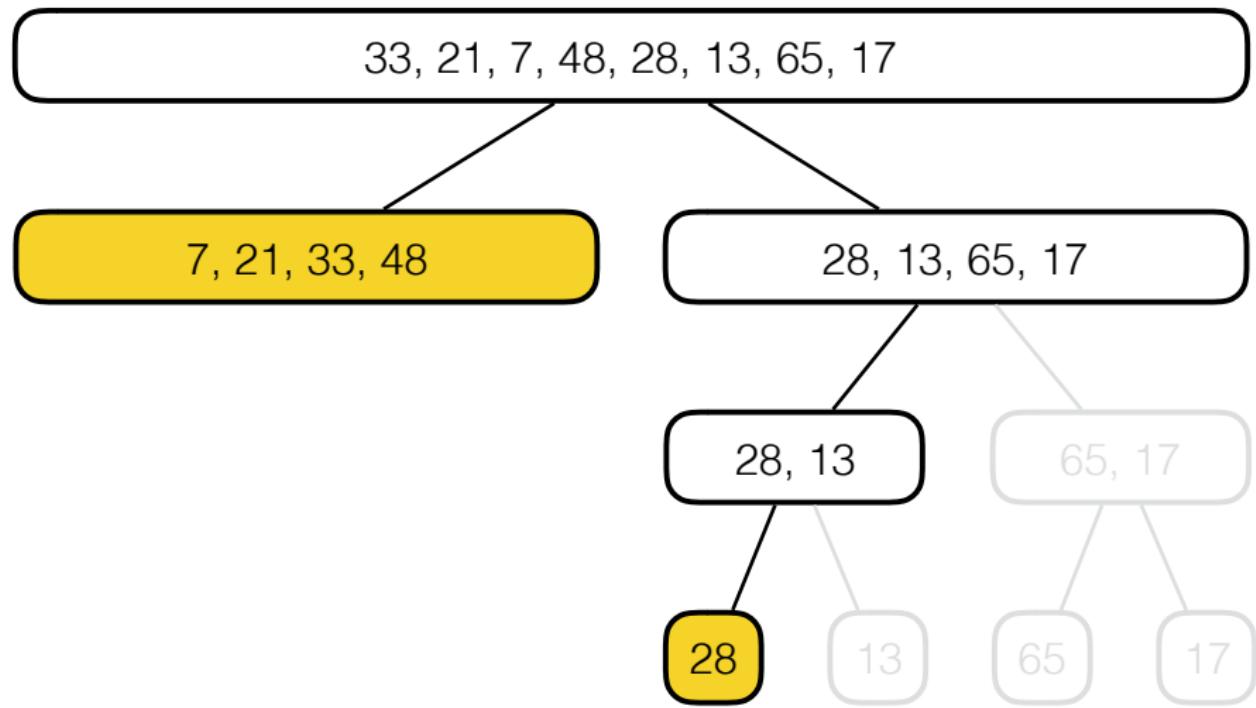
MergeSort(): Esecuzione



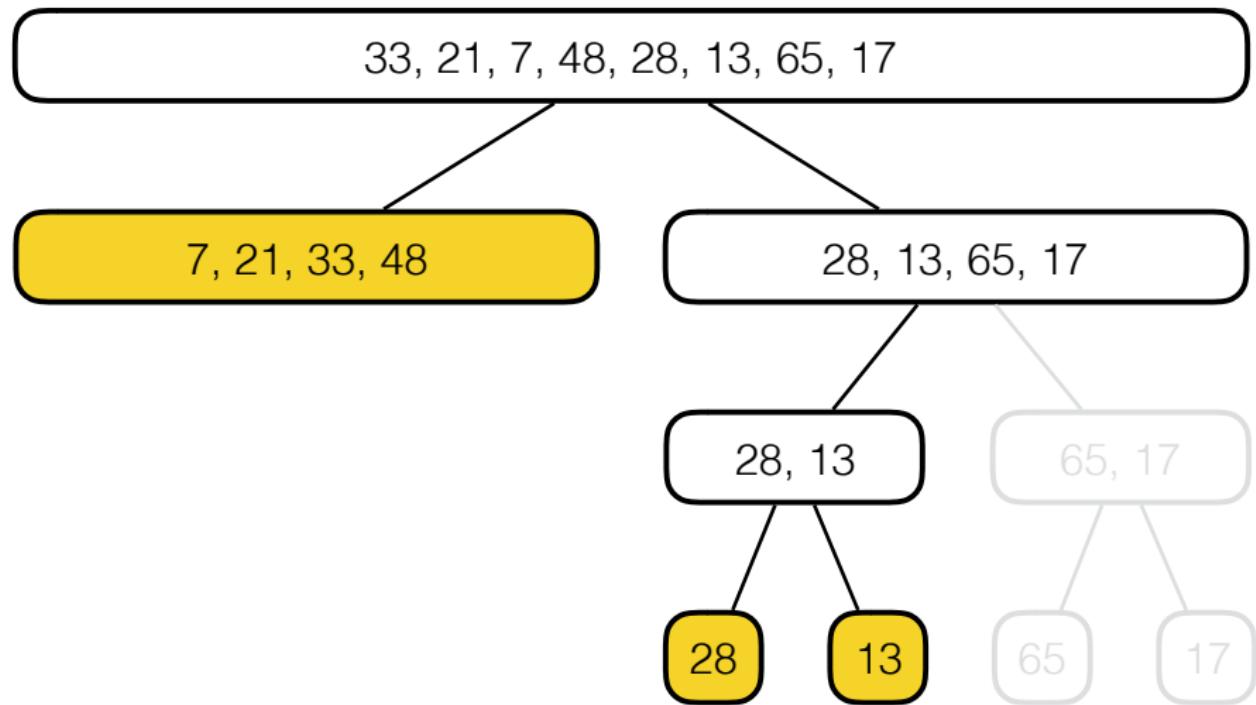
MergeSort(): Esecuzione



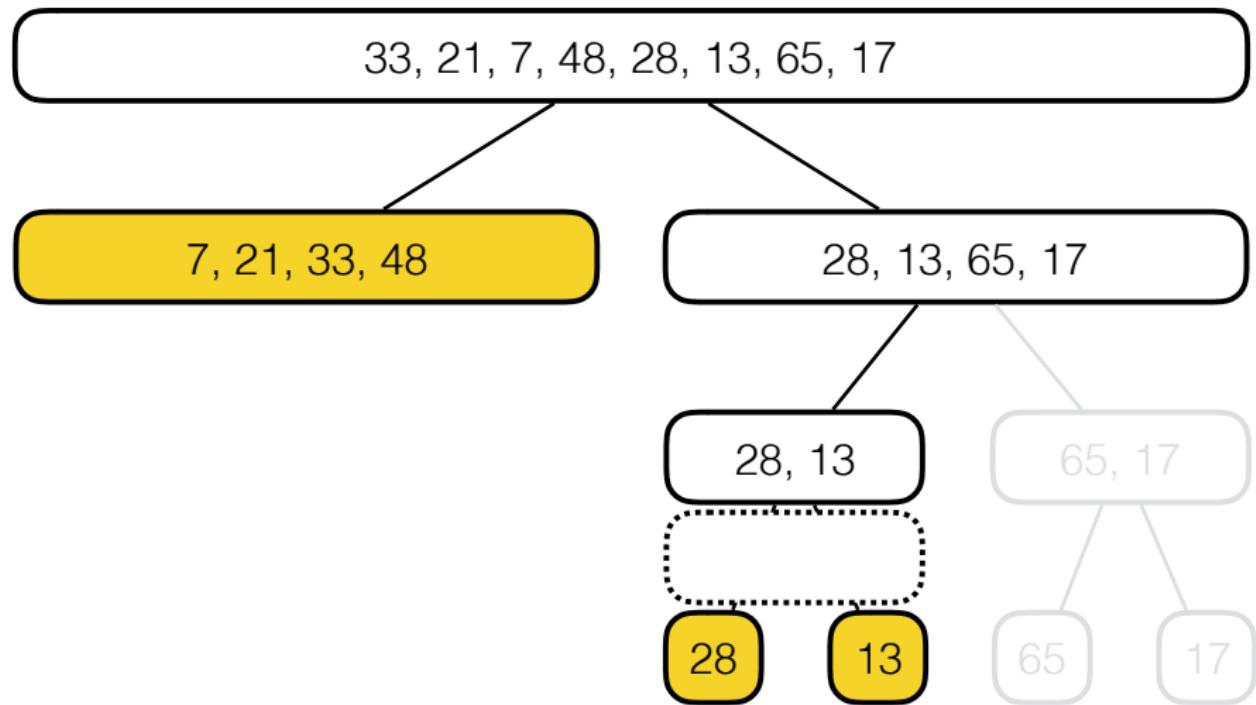
MergeSort(): Esecuzione



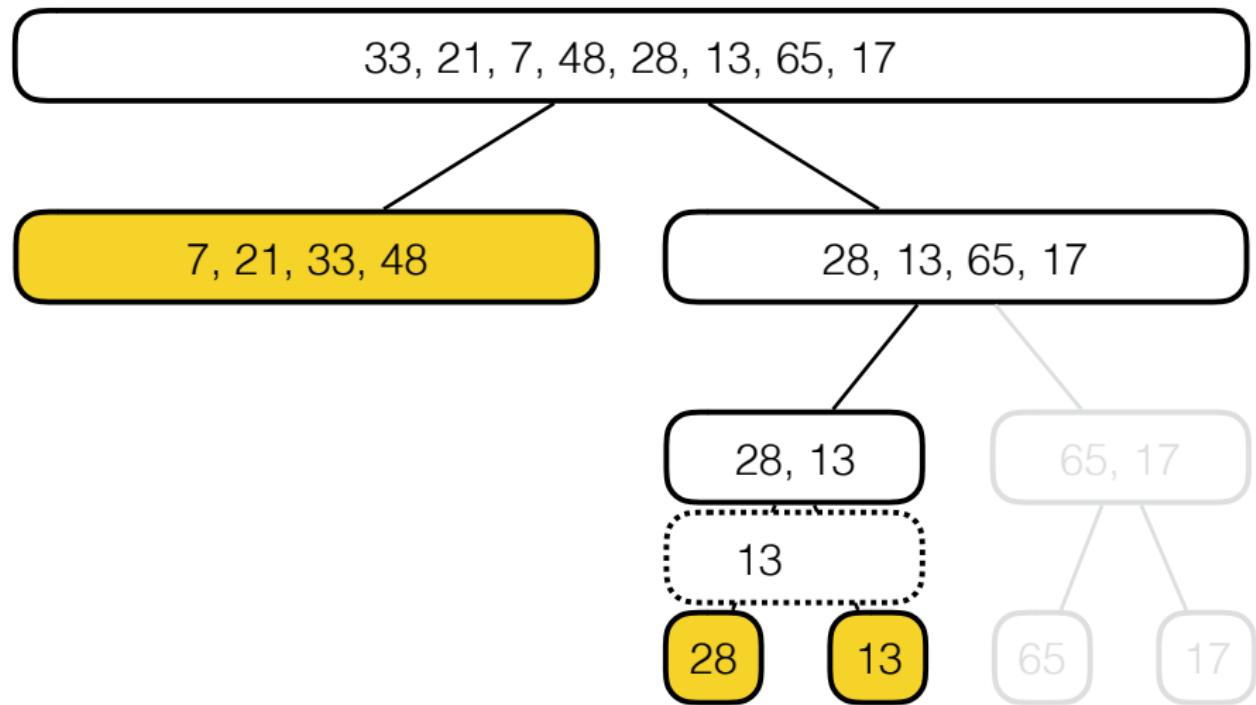
MergeSort(): Esecuzione



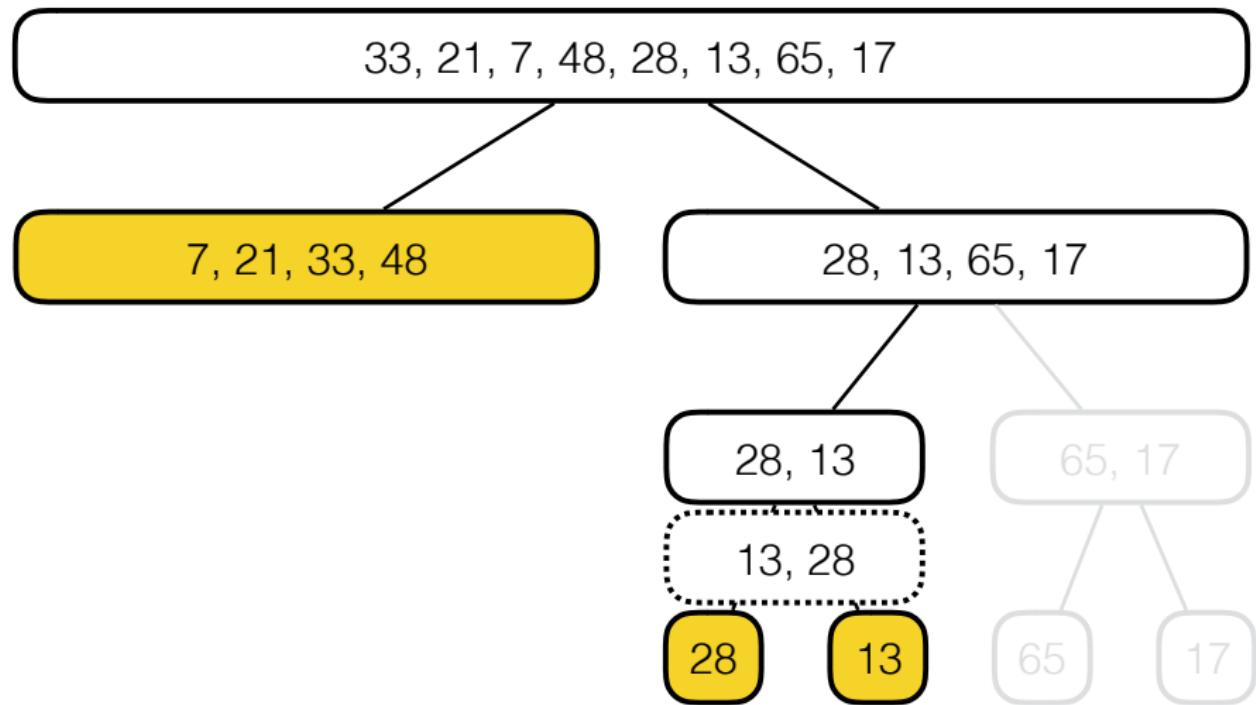
MergeSort(): Esecuzione



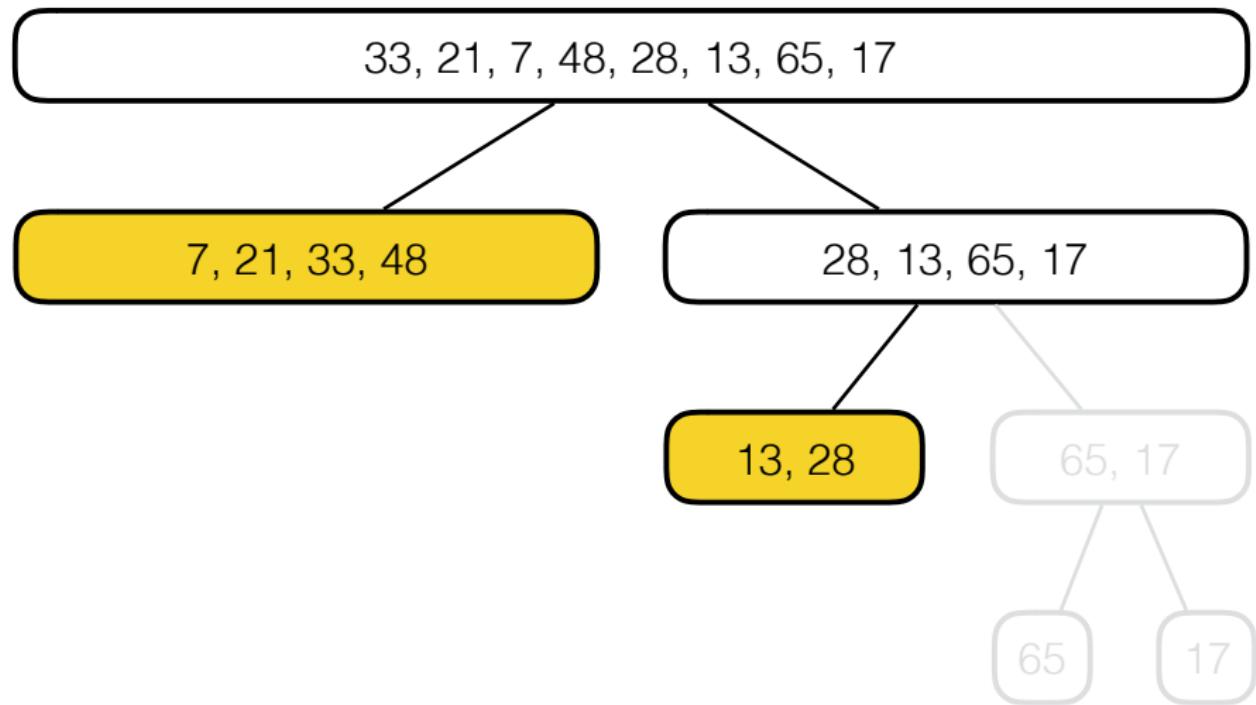
MergeSort(): Esecuzione



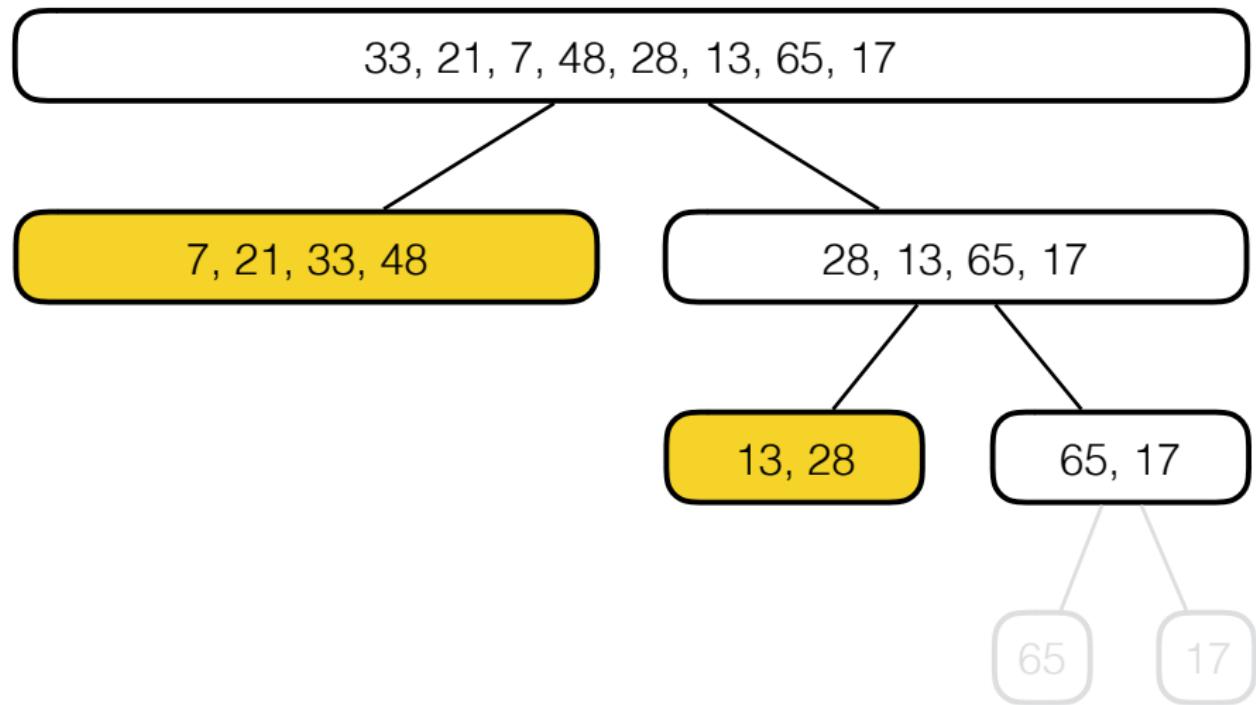
MergeSort(): Esecuzione



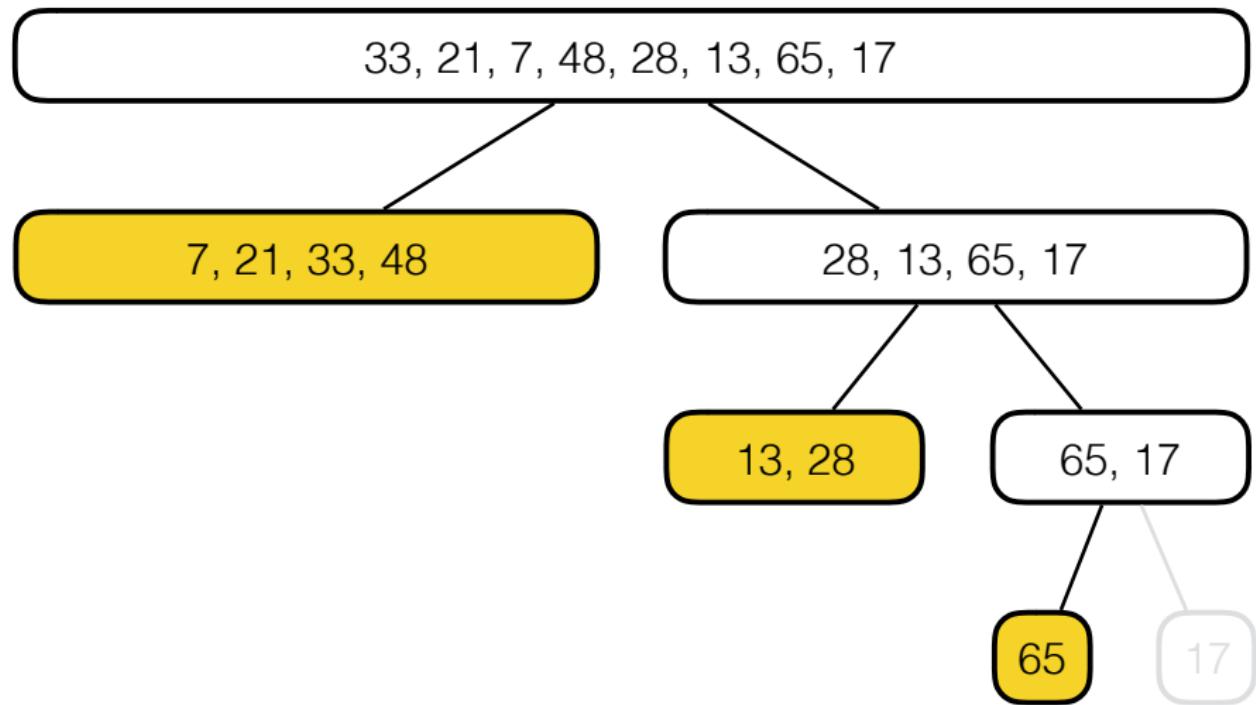
MergeSort(): Esecuzione



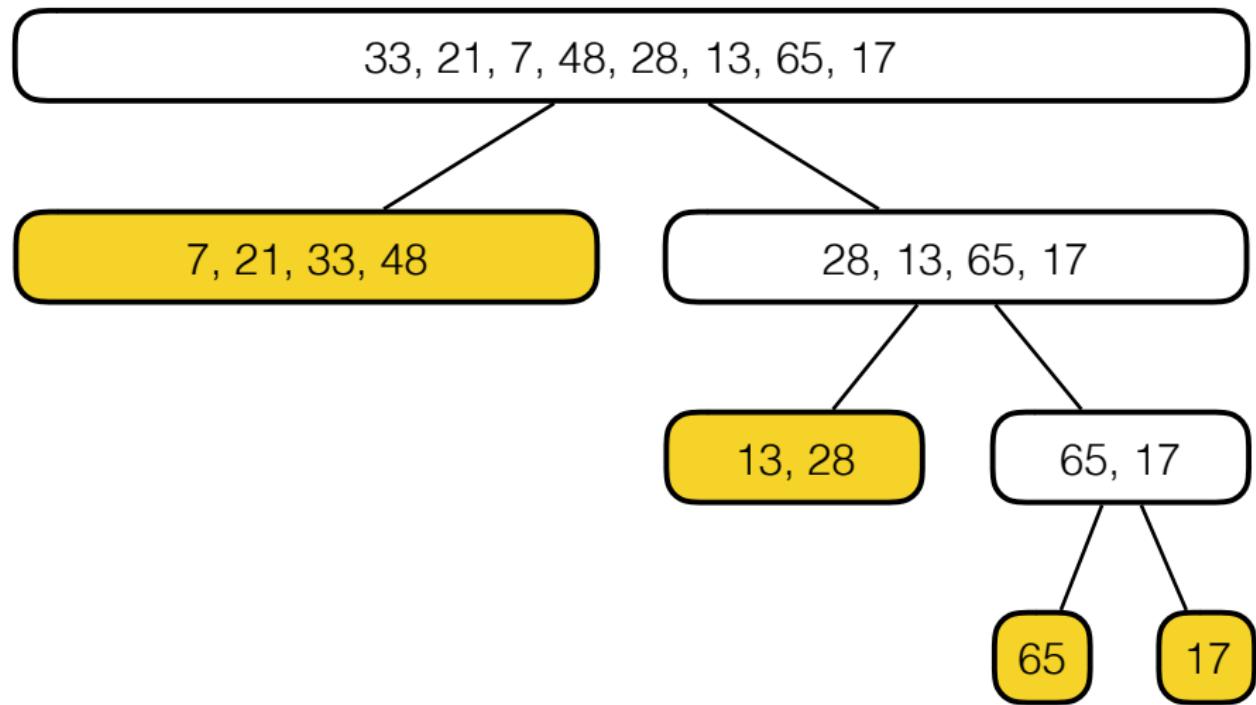
MergeSort(): Esecuzione



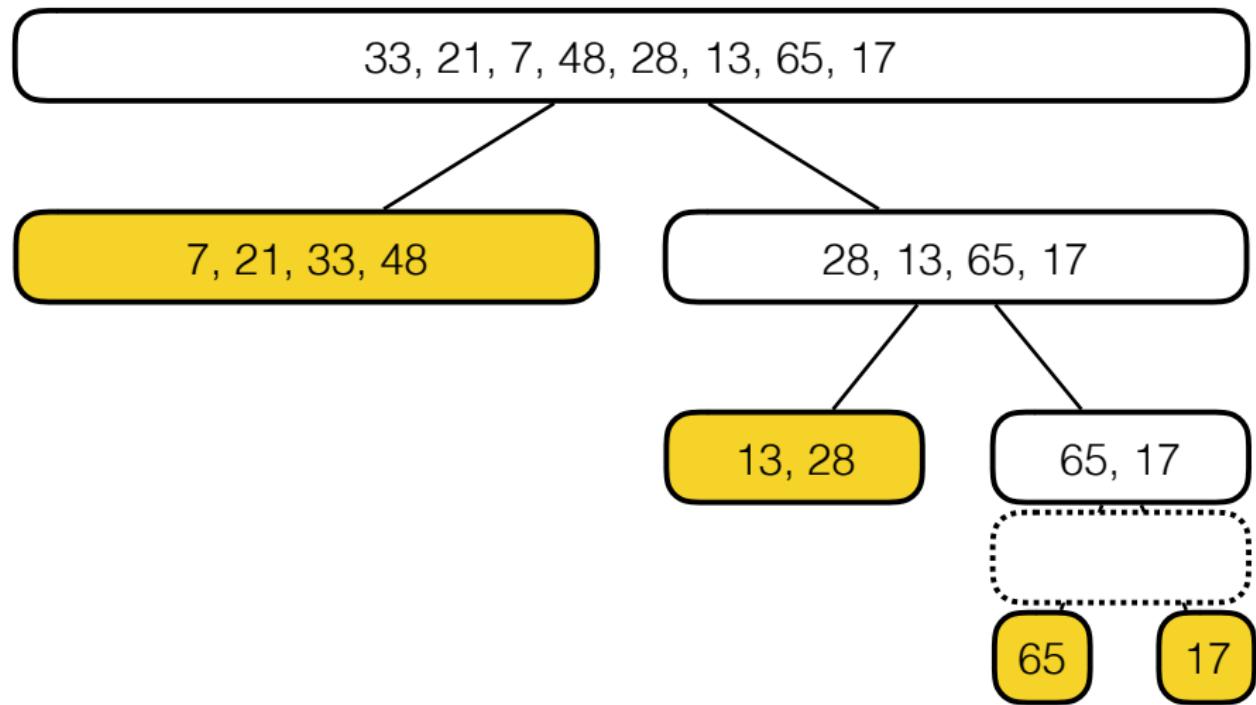
MergeSort(): Esecuzione



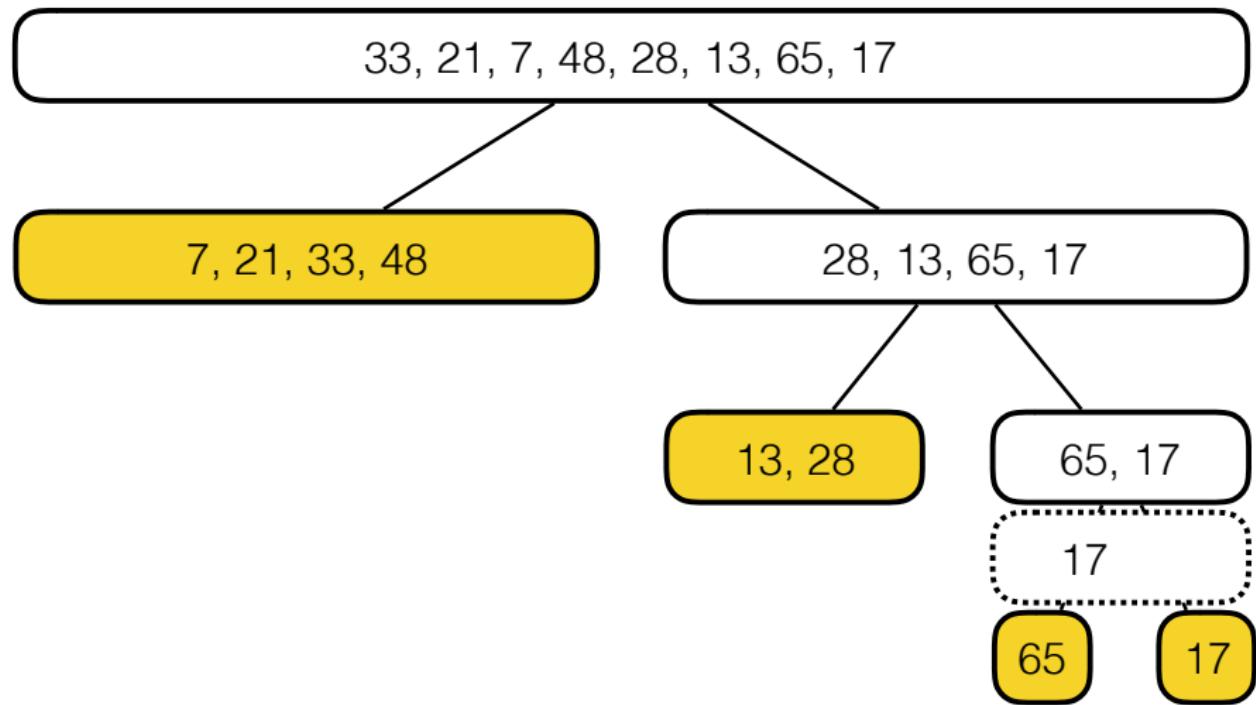
MergeSort(): Esecuzione



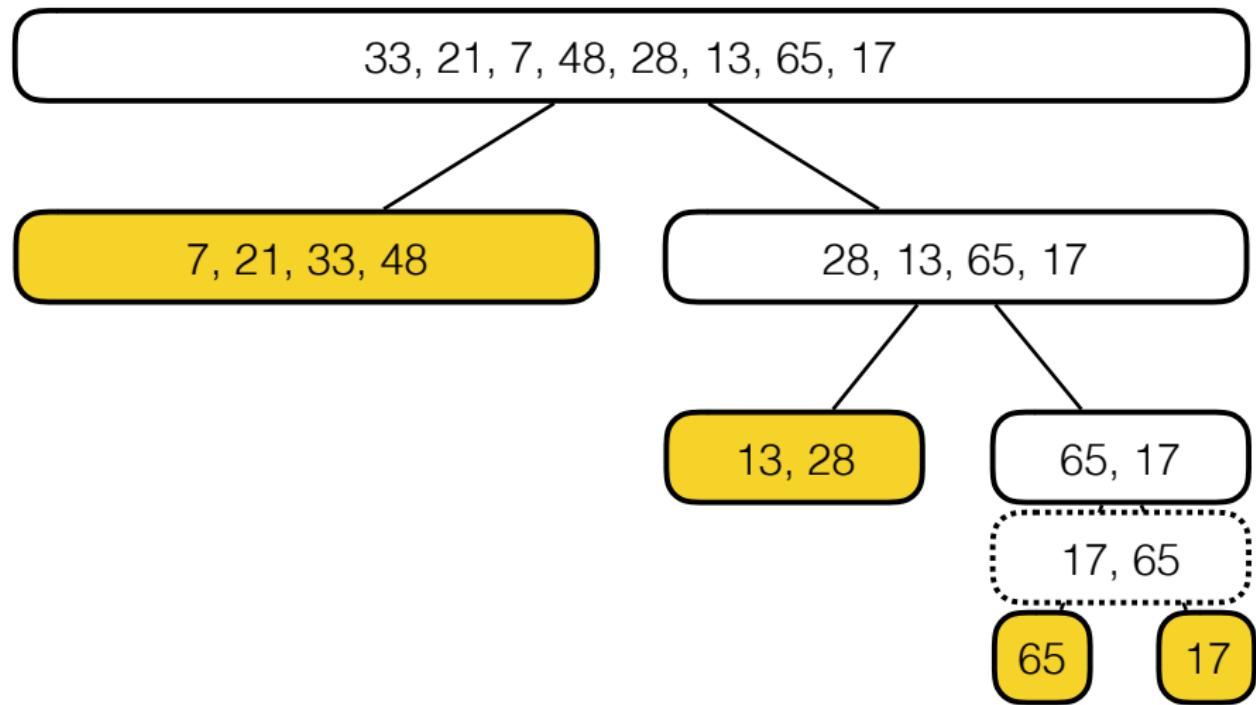
MergeSort(): Esecuzione



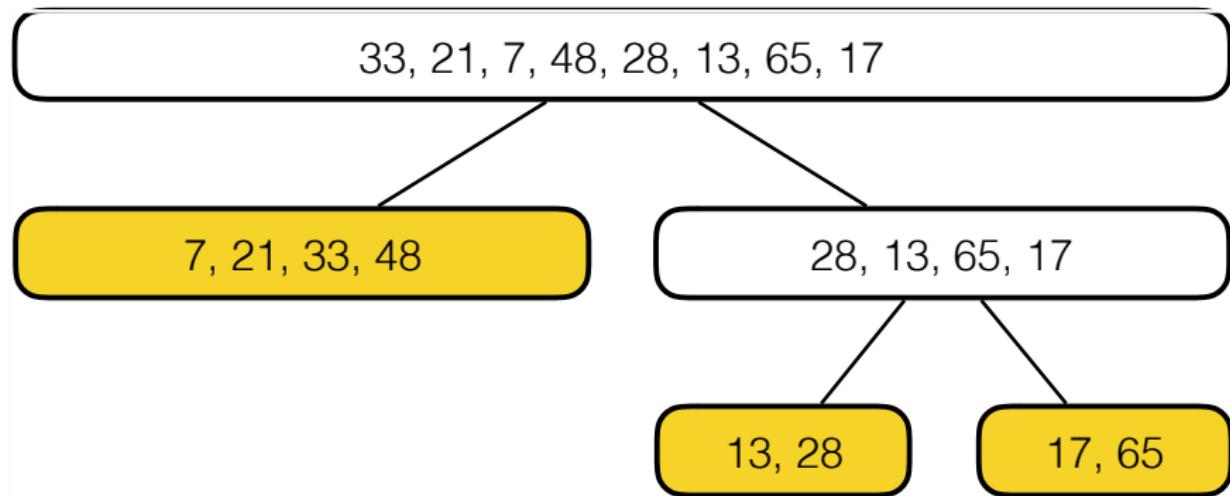
MergeSort(): Esecuzione



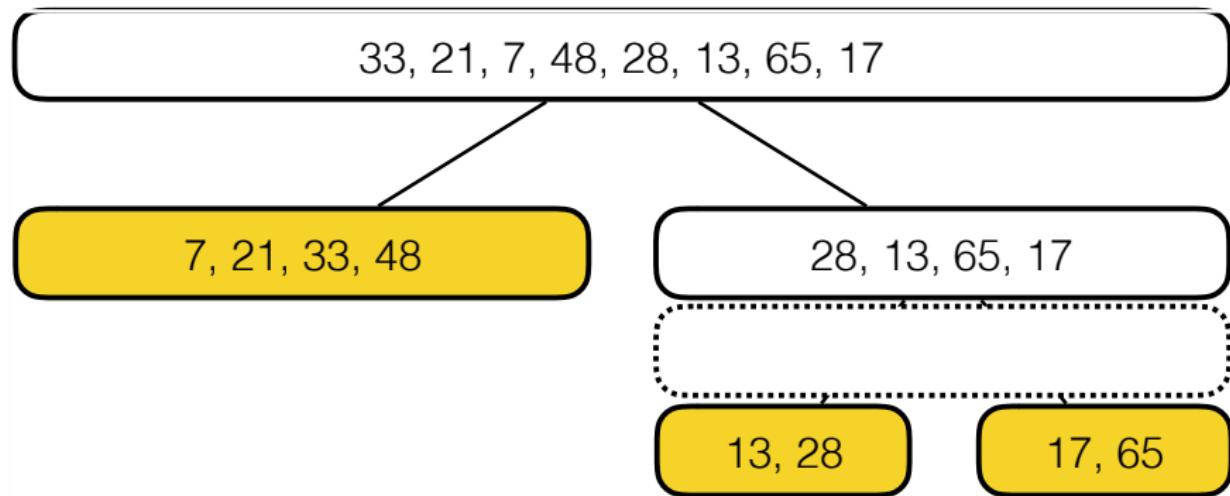
MergeSort(): Esecuzione



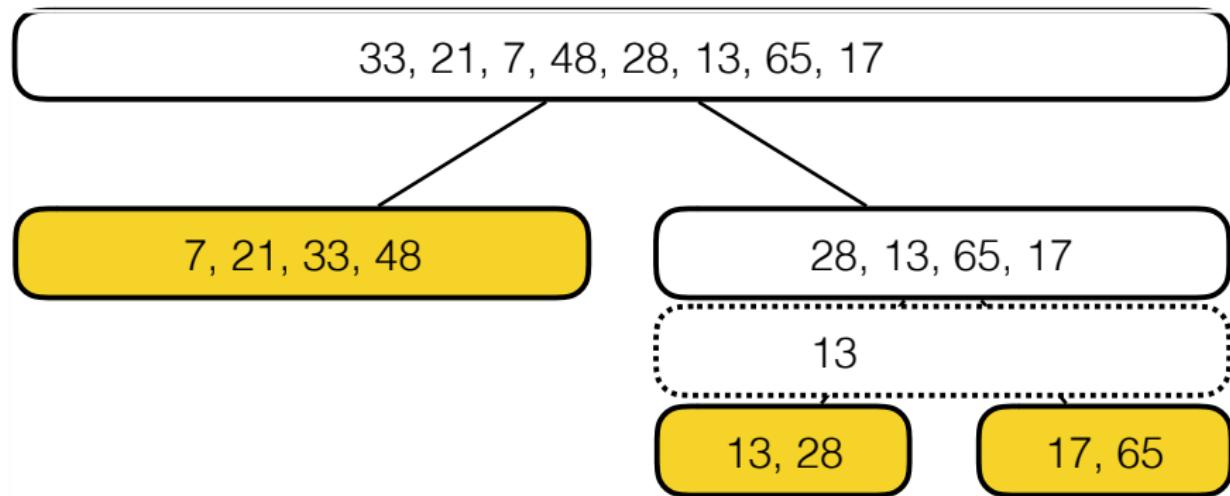
MergeSort(): Esecuzione



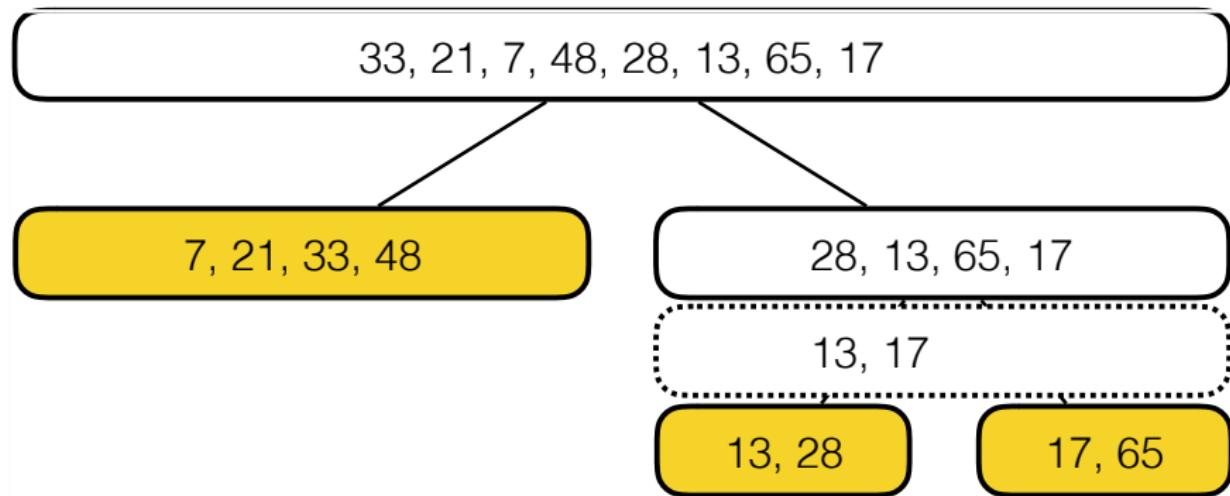
MergeSort(): Esecuzione



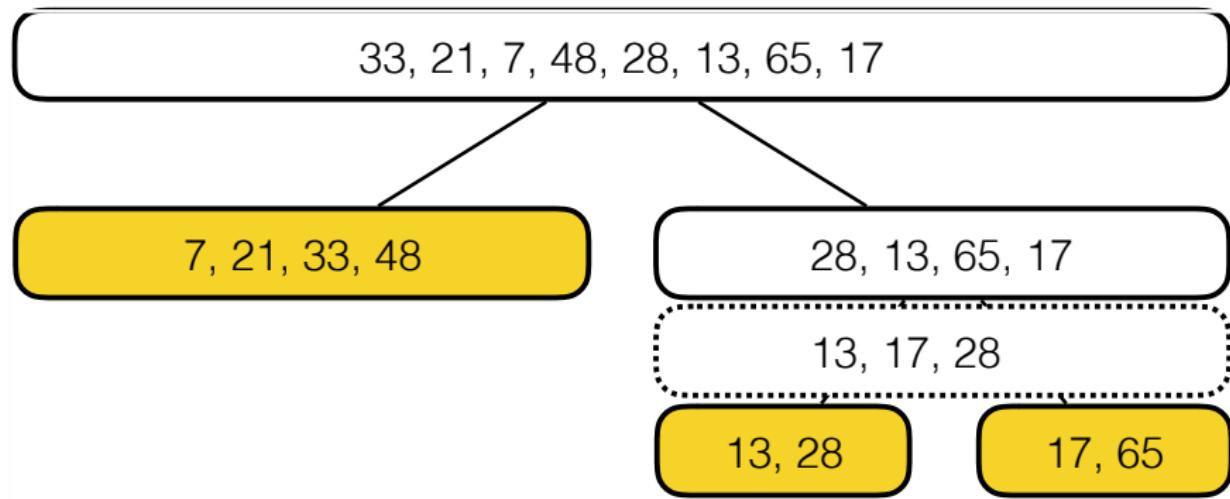
MergeSort(): Esecuzione



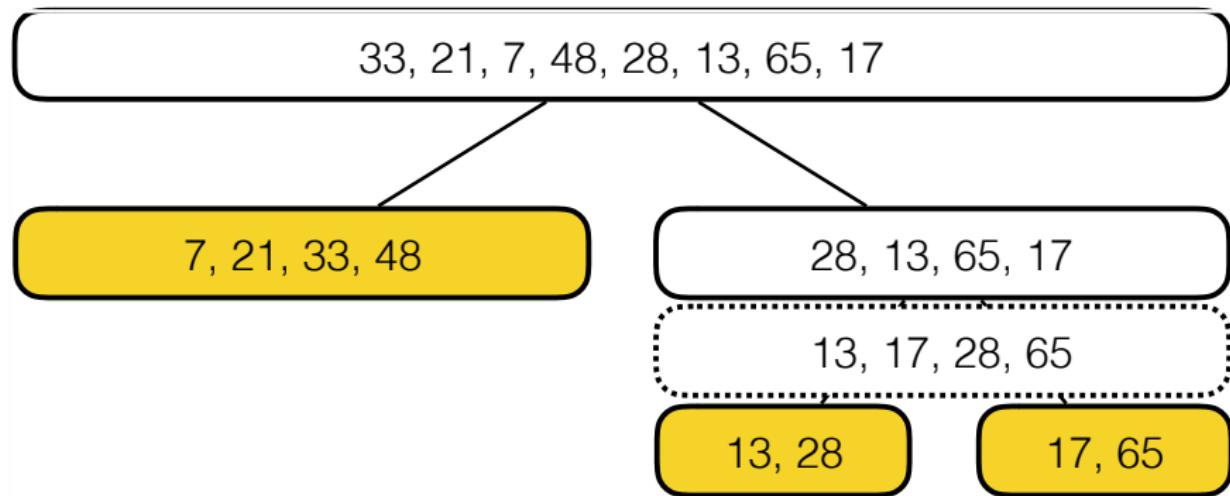
MergeSort(): Esecuzione



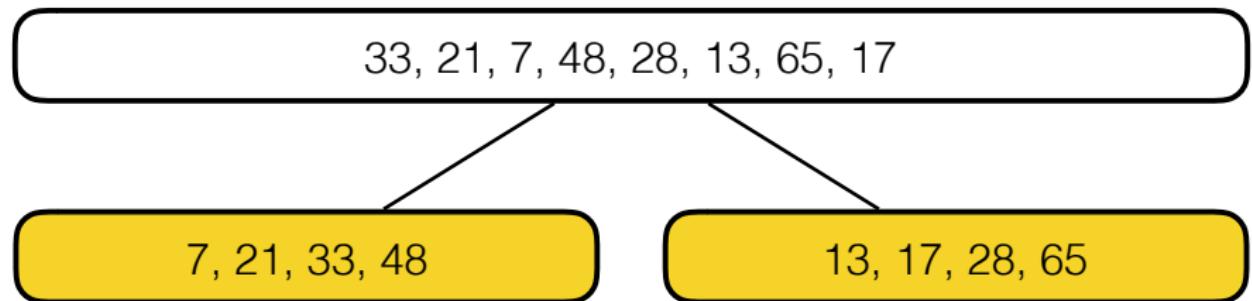
MergeSort(): Esecuzione



MergeSort(): Esecuzione



MergeSort(): Esecuzione



MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 13

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21, 28

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21, 28, 33

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21, 28, 33, 48

7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

33, 21, 7, 48, 28, 13, 65, 17

7, 13, 17, 21, 28, 33, 48, 65

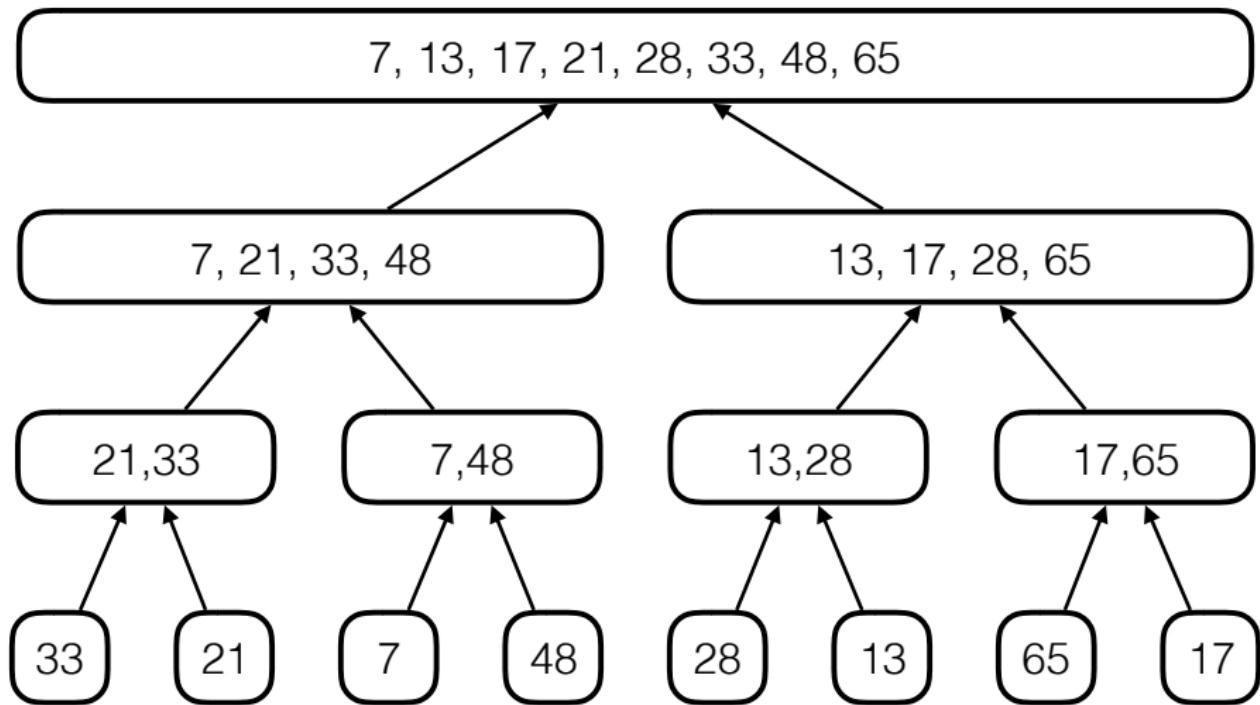
7, 21, 33, 48

13, 17, 28, 65

MergeSort(): Esecuzione

7, 13, 17, 21, 28, 33, 48, 65

MergeSort(): Esecuzione



Analisi di MergeSort()

Un'assunzione semplificativa:

- $n = 2^k$, ovvero l'altezza dell'albero di suddivisioni è esattamente $k = \log n$;
- Tutti i sottovettori hanno dimensioni che sono potenze esatte di 2

Costo computazionale

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

Analisi degli algoritmi del tipo divide et impera

$T(n) \rightarrow$ tempo di esecuzione di un problema di dimensione n nel caso peggiore

Se la dimensione del problema è piccola ($n < n_0$) per qualche costante n_0 possiamo assumere che la soluzione diretta richieda un tempo costante $\Theta(1)$

Supponiamo che la suddivisione del problema generi **a sottoproblemi**, ciascuno di dimensione n/b , cioè $1/b$ volte la dimensione del problema originale

⇒ Serve un tempo $T(n/b)$ per risolvere un sottoproblema di dimensioni n/b e quindi, per risolverne a serve un tempo $aT(n/b)$

Se impieghiamo un tempo $D(n)$ per dividere il problema in sottoproblemi e un tempo $C(n)$ per combinarne le soluzioni dei sottoproblemi nella soluzione del problema originale → Ottieniamo la ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq n_0 \\ aT(n/b) + D(n) + C(n) & \text{negli altri casi} \end{cases}$$

Analisi del Merge-Sort

Divide \rightarrow questo passo calcola semplicemente il centro del sottoarray. Ciò richiede un tempo costante, quindi $D(n) = \Theta(1)$

Impera \rightarrow si risolvono in modo ricorsivo i due sottoproblemi ciascuno di dimensione $n/2$, ciò contribuisce con $2T(n/2)$ al tempo di esecuzione

Combina \rightarrow poiché la procedura Merge su un sottoarray di n elementi richiede un tempo $\Theta(n)$, abbiamo $C(n) = \Theta(n)$

$$T(n) = 2T(n/2) + \Theta(n) \rightarrow \text{si risolve col Teorema Master} \Rightarrow T(n) = \Theta(n \log n)$$

Ricorrenza:

$$T(n) = \begin{cases} C_1 & \text{se } n=1 \\ 2T(n/2) + C_2 n & \text{se } n>1 \end{cases}$$

$C_1 \rightarrow$ tempo richiesto per risolvere un problema di dimensione 1

$C_2 \rightarrow$ tempo richiesto per gestire ogni singolo elemento nei passi divide e combina

Costo computazionale di Merge Sort

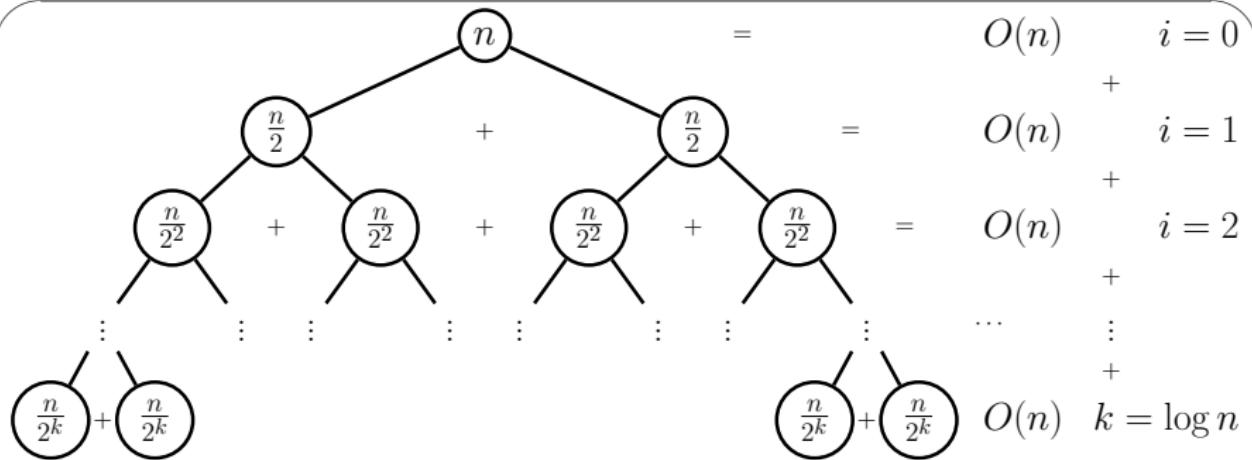
Domanda

Qual è il costo computazionale di `MergeSort()`?

Costo computazionale di Merge Sort

Domanda

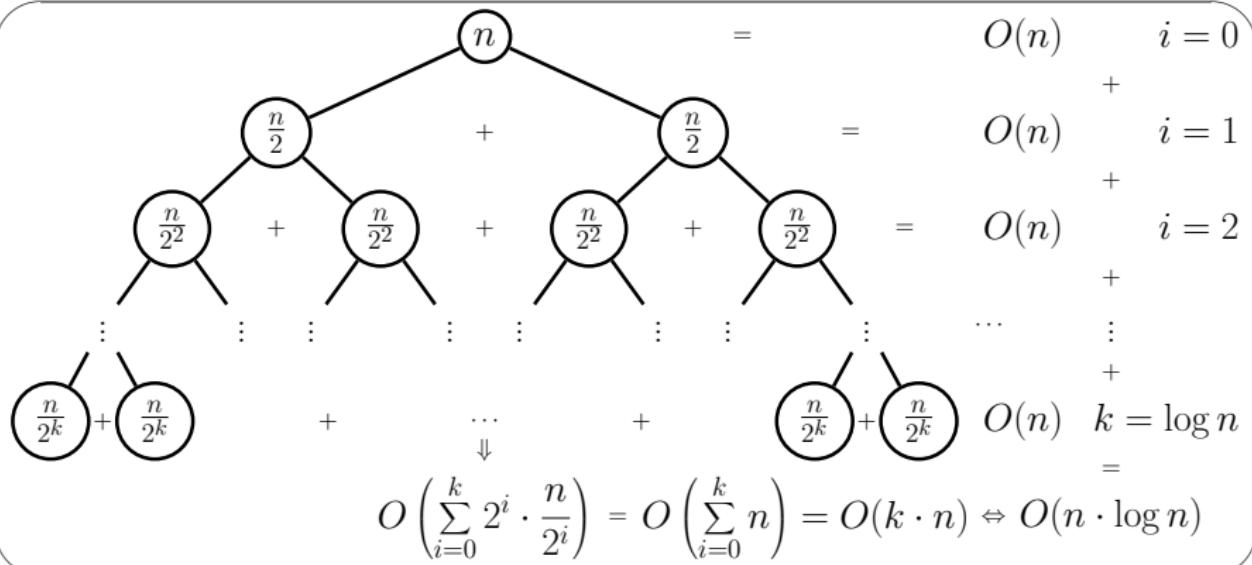
Qual è il costo computazionale di MergeSort()?



Costo computazionale di Merge Sort

Domanda

Qual è il costo computazionale di `MergeSort()`?



Un po' di storia

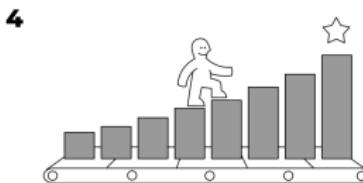
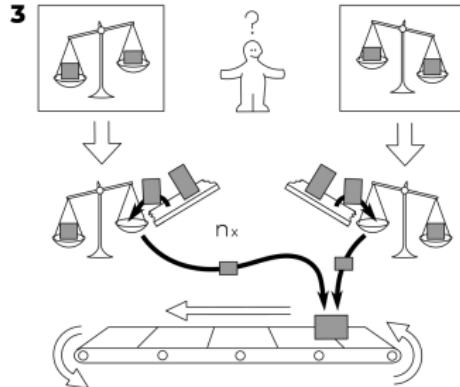
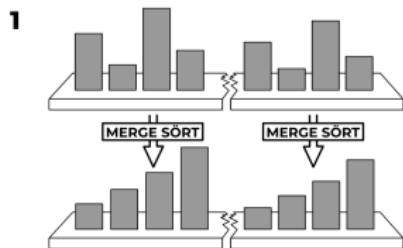
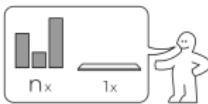
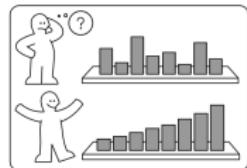
- Il censimento americano del 1880 aveva richiesto otto anni per essere completato
- Quello del 1890 richiese sei settimane, grazie alla Hollerith Machine
- Fra il 1896 e il 1924, la Hollerith & Co ha cambiato diversi nomi. L'ultimo?
International Business Machines
- Le Collating Machines (1936) prendevano due stack di schede perforate ordinate e le ordinavano in un unico stack
- Nel 1945-48, John von Neumann descrisse per la prima volta il MergeSort partendo dall'idea delle Collating Machines.



Hollerith Machine

MergeSort, in pillole

MERGE SÖRT



idea-instructions.com/merge-sort/
v1.1, CC by-nc-sa 4.0

Ricorsione

Recursion is the root of computation because it trades description for time.

Alan Perlis, Epigrams on Programming

A screenshot of a Google search results page. The search query 'recursion' is entered in the search bar. The results page shows a 'Search' section with a count of 'About 1,570,000 results (0.20 seconds)'. To the left, there is a sidebar with links to 'Everything', 'Images', 'Maps', 'Videos', 'News', and 'More'. Below the sidebar, there is a link to 'Pune, Maharashtra' and 'Channel location'. A red oval highlights the 'Did you mean: recursion' suggestion above the first search result. The first result is a link to 'Recursion - Wikipedia, the free encyclopedia' with the URL 'en.wikipedia.org/wiki/Recursion'. The snippet for this result describes recursion as a process of repeating items in a self-similar way. The second result is a link to 'Recursion (computer science) - Wikipedia, the free encyclopedia' with the URL 'en.wikipedia.org/wiki/Recursion_(computer_science)'. The snippet for this result describes recursion in computer science as a method where the solution to a problem depends on solutions to smaller instances of the same problem.

+Vishal Web Images Maps News Orkut Gmail More ▾

Google recursion

Search About 1,570,000 results (0.20 seconds)

Everything Did you mean: *recursion*

Images

Maps

Videos

News

More

Pune, Maharashtra
Channel location

[Recursion - Wikipedia, the free encyclopedia](#)
en.wikipedia.org/wiki/Recursion
Recursion is the process of repeating items in a self-similar way
the surfaces of two mirrors are exactly parallel with each other th
Formal definitions of recursion - Recursion in language - Recursio

[Recursion \(computer science\) - Wikipedia, the free encyclopedia](#)
[en.wikipedia.org/wiki/Recursion_\(computer_science\)](http://en.wikipedia.org/wiki/Recursion_(computer_science))
Recursion in computer science is a method where the solution t
Recursive data types - Recursive algorithms - Structural versus g