

Algoritmi e Strutture Dati

Analisi di algoritmi Introduzione

Alberto Montresor

Università di Trento

2020/09/22

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 complessità
- 2 Notazione asintotica
 - Definizioni
 - Esercizi
- 3 Complessità problemi vs algoritmi
 - Moltiplicare numeri complessi
 - Sommare numeri binari
 - Moltiplicare numeri binari
- 4 Tipologia dell'input
 - Selection Sort
 - Insertion Sort
 - Merge Sort

Introduzione

Obiettivo: **stimare la complessità in tempo**

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

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

Complessità

Complessità: "**Dimensione dell'input**" \rightarrow "**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` ?
- `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"

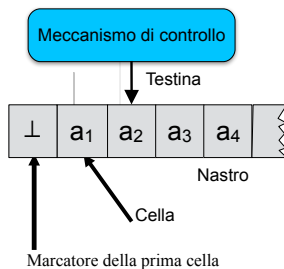
The following 108 pages are in this category, out of 108 total. This list may not reflect recent changes ([learn more](#)).

	E cont.	P cont.
A	<ul style="list-style-type: none"> Model of computation 	<ul style="list-style-type: none"> Probabilistic Turing machine Pushdown automaton
<ul style="list-style-type: none"> Abstract Job Object Abstract machine Abstract state machines Agent-based model Algorithm characterizations Alternating Turing machine Applicative computing systems 	F	Q
	<ul style="list-style-type: none"> Event-driven finite-state machine Evolution in Variable Environment Extended finite-state machine 	<ul style="list-style-type: none"> Quantum capacity Quantum circuit Quantum computer
R	<ul style="list-style-type: none"> Finite state machine with datapath Finite state transducer Finite-state machine FRACTRAN Funnelsort 	R
		<ul style="list-style-type: none"> 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, ininfluenza 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 $\min(\text{ITEM[]} A, \text{int } n)$

	Costo	# Volte
ITEM $\text{min} = A[1]$	c_1	1
for $i = 2$ to n do	c_2	n
if $A[i] < \text{min}$ then	c_3	$n - 1$
$\text{min} = A[i]$	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	# ($i > j$)	# ($i \leq j$)
if $i > j$ then	c_1	1	1
return 0	c_2	1	0
else			
int $m = \lfloor (i+j)/2 \rfloor$	c_3	0	1
if $A[m] = v$ then	c_4	0	1
return m	c_5	0	0
else if $A[m] < v$ then	c_6	0	1
return <code>binarySearch</code> ($A, v, m+1, j$)	$c_7 + T(\lfloor (n-1)/2 \rfloor)$	0	0/1
else			
return <code>binarySearch</code> ($A, v, i, m-1$)	$c_7 + T(\lfloor n/2 \rfloor)$	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:

$$i > j \quad (n = 0)$$

$$T(n) = c_1 + c_2 = c$$

$$i \leq j \quad (n > 0)$$

$$\begin{aligned} T(n) &= T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7 \\ &= T(n/2) + d \end{aligned}$$

- Relazione di ricorrenza:

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

Tempo di calcolo `binarySearch()`

Soluzione della relazione di ricorrenza tramite espansione

$$\begin{aligned}T(n) &= T(n/2) + d \\&= T(n/4) + 2d \\&= T(n/8) + 3d \\&\dots \\&= T(1) + kd \\&= T(0) + (k+1)d \\&= kd + (c+d) \\&= d \log n + e.\end{aligned}$$

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

Ordini di complessità

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

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

Ordini di complessità

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

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

Ordini di complessità

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

- 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** $O(n^2)$

Classi di complessità

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	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

Università di Trento

2020/09/22

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



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 $g(n)$

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 $g(n)$

Notazioni O , Ω , Θ

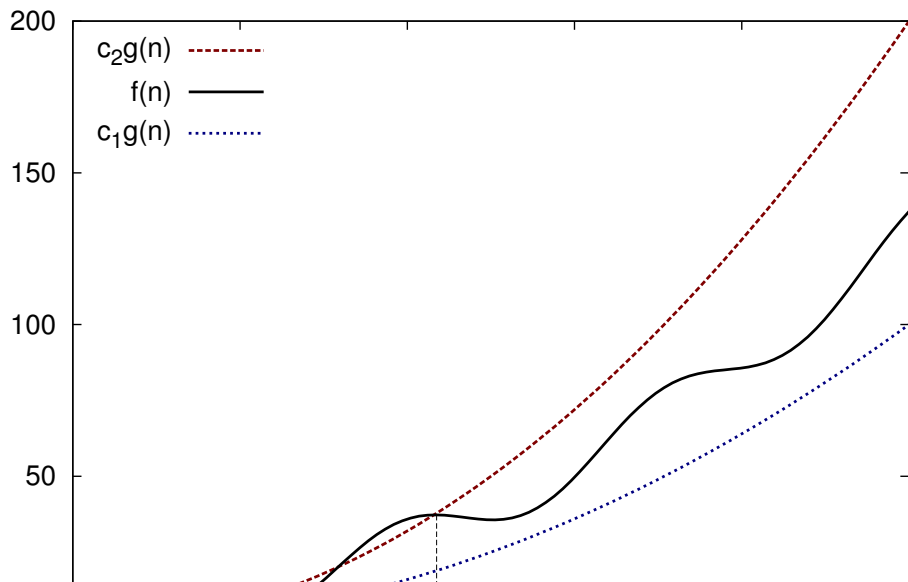
Definizione – Notazione Θ

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 $f(n) = \Omega(g(n))$

Graficamente



Vero o falso?

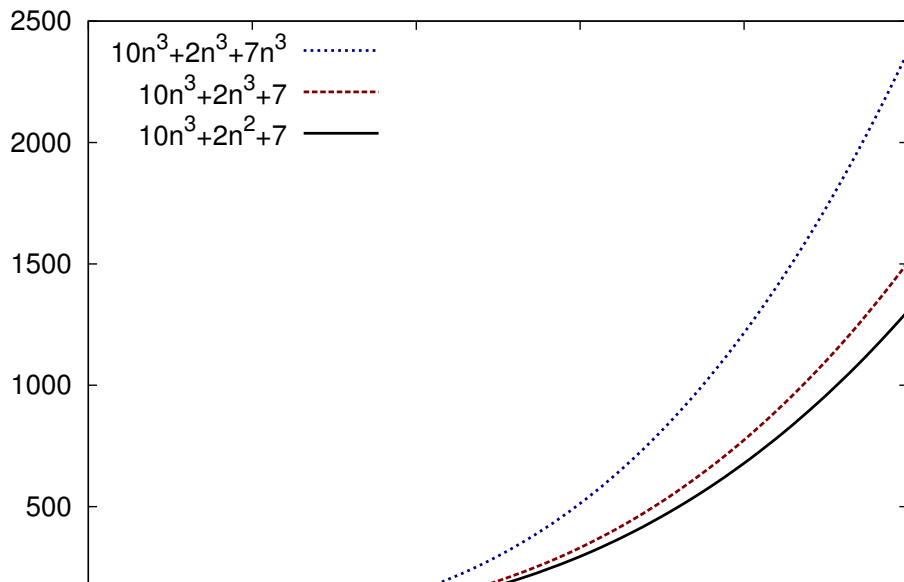
$$f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} O(n^3)$$

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

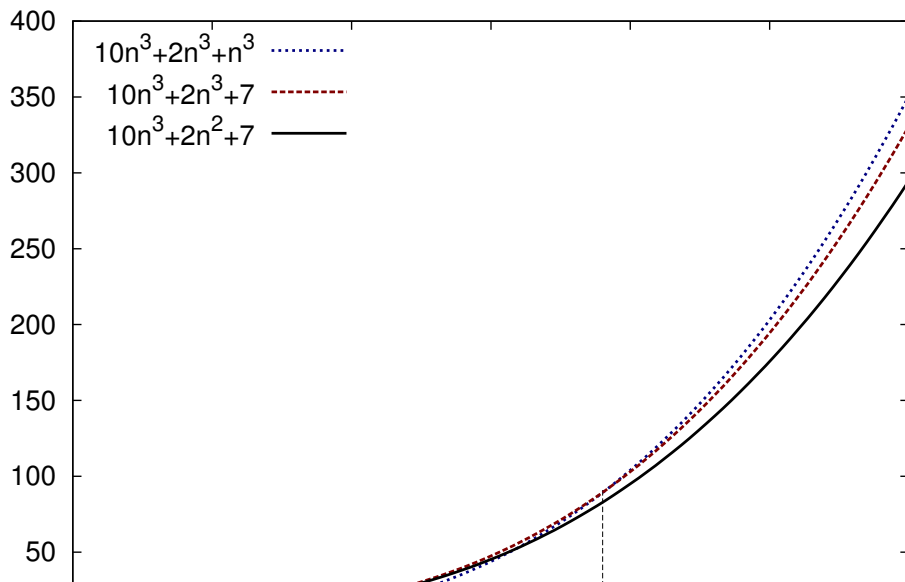
$$f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} O(n^3)$$

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

$$f(n) = 3n^2 + 7n$$

$$\geq 3n^2$$

Per $n \geq 0$

$$\stackrel{?}{\geq} c_1 n^2$$

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 :

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

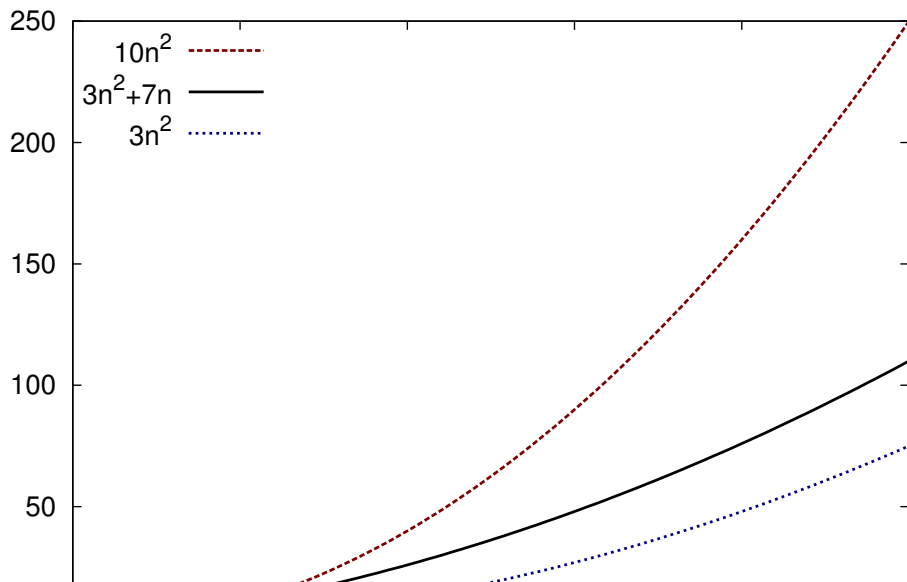
Con questi parametri:

$$c_1 = 3$$

$$c_2 = 10$$

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

Graficamente



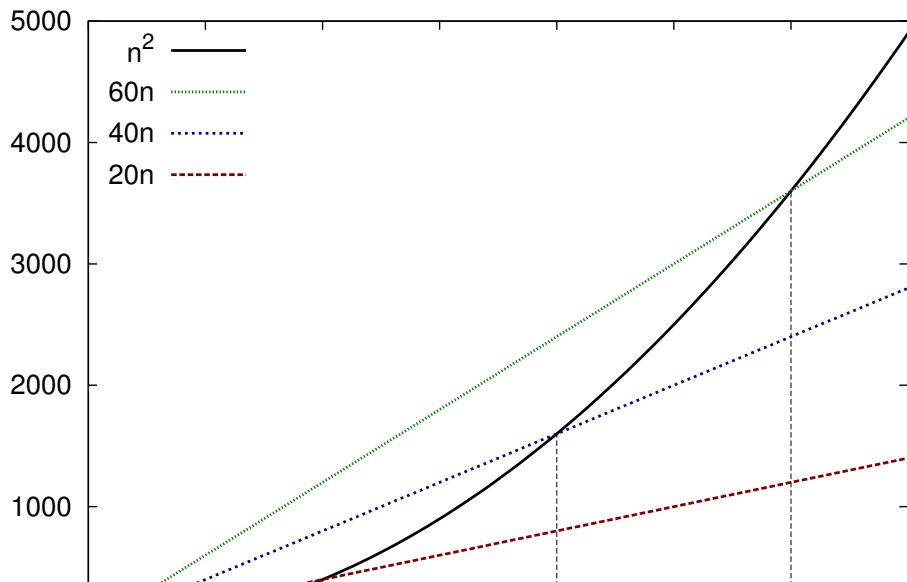
Vero o falso?

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

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

Complessità algoritmi vs Complessità problemi

Alberto Montresor

Università di Trento

2020/09/22

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



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

Moltiplicare numeri complessi

Moltiplicazione numeri complessi

- $(a + bi)(c + di) = [ac - bd] + [ad + bc]i$
- Input: a, b, c, d
- Output: $ac - bd, ad + bc$

Domande

Considerate un modello di calcolo dove la moltiplicazione costa 1, le addizioni/sottrazioni costano 0.01.

- Quanto costa l'algoritmo dettato dalla definizione?
- Potete fare meglio di così?
- Qual è il ruolo del modello di calcolo?

Moltiplicare numeri complessi

Questioni aperte

- Si può fare ancora meglio?
- Oppure, è possibile dimostrare che non si può fare meglio di così?

Alcune riflessioni

- In questo modello, effettuare 3 moltiplicazioni invece di 4 risparmia il 25% del costo
- Esistono contesti in cui effettuare 3 moltiplicazioni invece di 4 può produrre un risparmio maggiore

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?

$$\begin{array}{cccccccccccccccc}
 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow & \swarrow \\
 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
 \hline
 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1
 \end{array}$$

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 complessità $O(f(n))$

Limite superiore della somma di numeri binari

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

Limite inferiore alla complessità di un problema

Notazione $\Omega(f(n))$ – Limite inferiore

Un problema ha complessità $\Omega(f(n))$ se tutti i possibili algoritmi che lo risolvono hanno complessità $\Omega(f(n))$.

Limite inferiore della somma di numeri binari

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

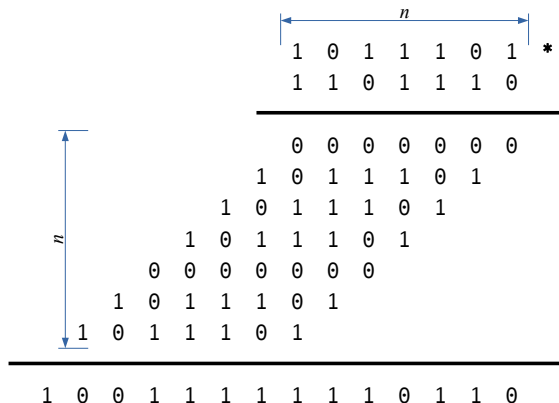
Domanda

Riuscite a dimostrarlo?

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 inerentemente 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!

Moltiplicare numeri binari

Divide-et-impera

- **Divide**: dividi il problema in sottoproblemi di dimensioni inferiori
- **Impera**: risolvi i sottoproblemi in maniera ricorsiva
- **Combina**: unisci le soluzioni dei sottoproblemi in modo da ottenere la risposta del problema principale

Moltiplicazione divide-et-impera

$$X = a \cdot 2^{n/2} + b$$

$$Y = c \cdot 2^{n/2} + d$$

$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$

X	a	b
Y	c	d

Moltiplicare numeri binari tramite Divide-et-impera

```
boolean [] pdi(boolean[] X, boolean[] Y, int n)
```

```
if n == 1 then
```

```
    return X[1] · Y[1]
```

```
else
```

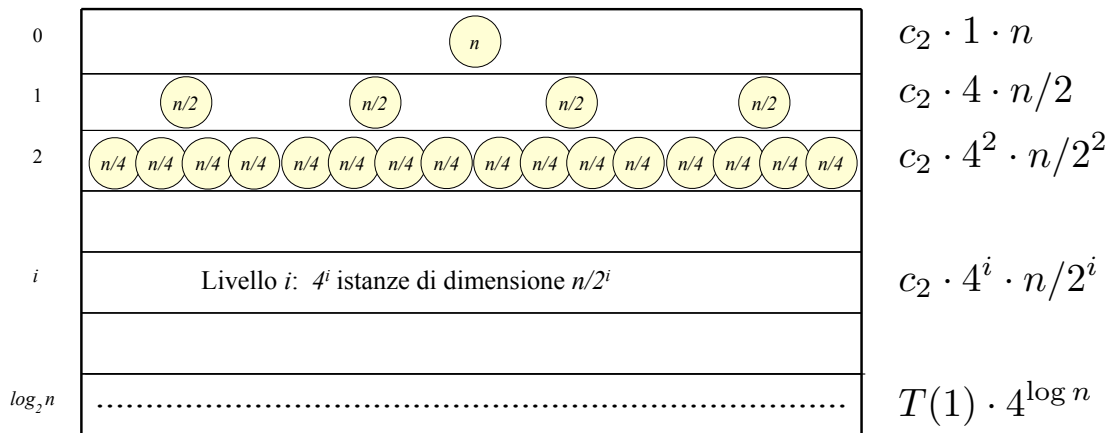
```
    spezza X in a; b e Y in c; d
```

```
    return pdi(a, c, n/2) · 2n + (pdi(a, d, n/2) +  
        pdi(b, c, n/2)) · 2n/2 + pdi(b, d, n/2)
```

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Nota: Moltiplicare per $2^t \equiv$ shift di t posizioni, in tempo lineare

Analisi della ricorsione



$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

$$= c_1 \cdot n^{\log 4}$$

$$= c_1 \cdot n^2$$

Moltiplicare numeri binari

Confronto della complessità computazionale

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

Domanda: Tutto questo lavoro per nulla?

Non solo la complessità è uguale, ma le costanti moltiplicative sono più alte.

Domanda: E' possibile fare meglio di così?

Notate che la versione ricorsiva chiama se stessa 4 volte.

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] · Y[1]
```

```
else
```

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

```
    boolean[] A1 = KARATSUBA(a, c, n/2)
```

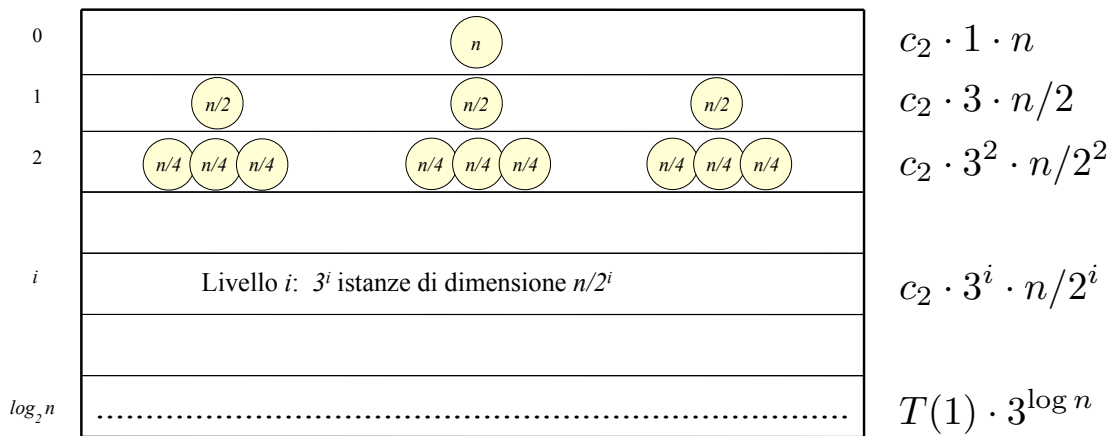
```
    boolean[] A3 = KARATSUBA(b, d, n/2)
```

```
    boolean[] m = KARATSUBA(a + b, c + d, n/2)
```

```
    boolean[] A2 = m - A1 - A3
```

```
    return A1 · 2n + A2 · 2n/2 + A3
```

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 : $T_{kara}(n) = O(n^{1.58...})$ 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)
 - 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

Complessità 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 relative 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

Università di Trento

2020/09/22

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



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 $b_1 \leq b_2 \leq \dots \leq b_n$

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

```
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  $min$ 
```

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
```

	<i>j</i> = 1	<i>j</i> = 2	<i>j</i> = 3	<i>j</i> = 4	<i>j</i> = 5	<i>j</i> = 6	<i>j</i> = 7
<i>i</i> = 1	7	4	2	1	8	3	5
<i>i</i> = 2	1	4	2	7	8	3	5
<i>i</i> = 3	1	2	4	7	8	3	5
<i>i</i> = 4	1	2	3	7	8	4	5
<i>i</i> = 5	1	2	3	4	8	7	5
<i>i</i> = 6	1	2	3	4	5	7	8
<i>i</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] ↔ 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?

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

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
```

```
    ITEM  $temp = A[i]$ 
```

```
    int  $j = i$ 
```

```
    while  $j > 1$  and  $A[j - 1] > temp$  do
```

```
         $A[j] = A[j - 1]$ 
```

```
         $j = j - 1$ 
```

```
     $A[j] = temp$ 
```

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

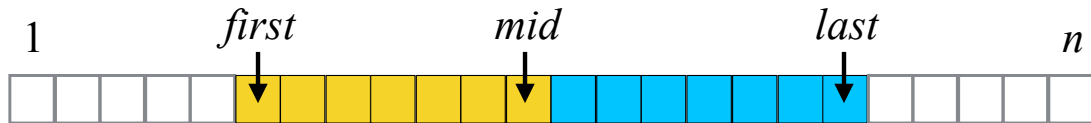
Merge Sort è basato sulla tecnica **divide-et-impera** vista in precedenza

- **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

Merge



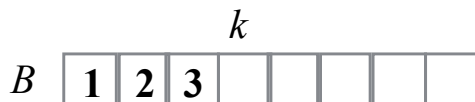
Input:

- 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

$i = first$

$j = mid + 1$

$k = first$

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

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

$B[k] = A[i]$

$i = i + 1$

else

$B[k] = A[j]$

$j = j + 1$

$k = k + 1$

$j = last$

for $h = mid$ **downto** i **do**

$A[j] = A[h]$

$j = j - 1$

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

$A[j] = B[j]$

Costo computazionale

Domanda

Qual è il costo computazionale di Merge()?

Costo computazionale

Domanda

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

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
```

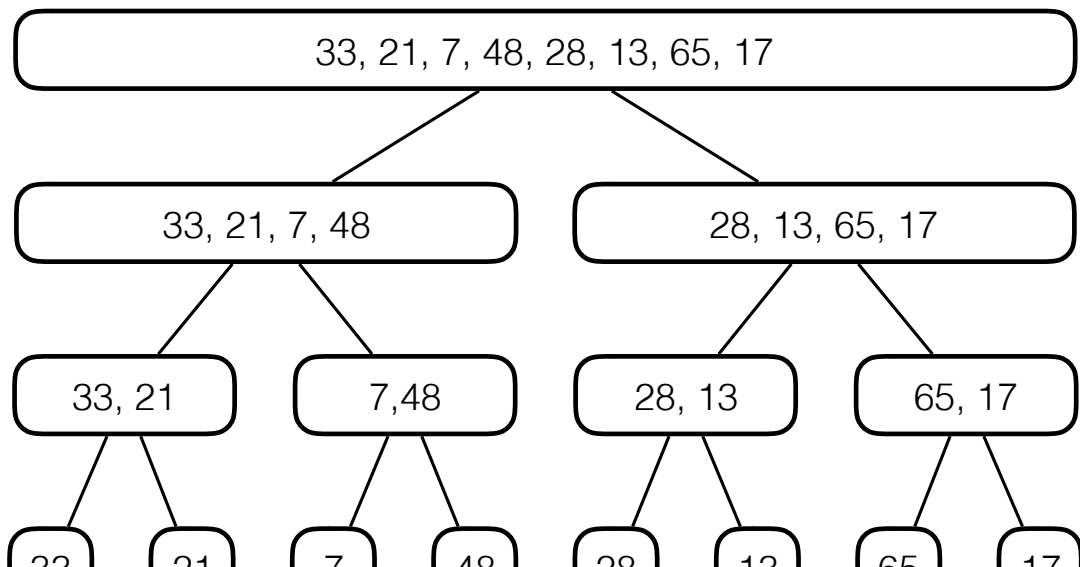
```
    int mid =  $\lfloor (first + last) / 2 \rfloor$ 
```

```
    MergeSort(A, first, mid)
```

```
    MergeSort(A, mid + 1, last)
```

```
    Merge(A, first, last, mid)
```

MergeSort(): Esecuzione



MergeSort(): Esecuzione

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

33, 21, 7, 48

28, 13, 65, 17

33, 21

7, 48

28, 13

65, 17

33

21

7

48

28

13

65

17

MergeSort(): Esecuzione

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

33, 21, 7, 48

28, 13, 65, 17

33, 21

7, 48

28, 13

65, 17

33

21

7

48

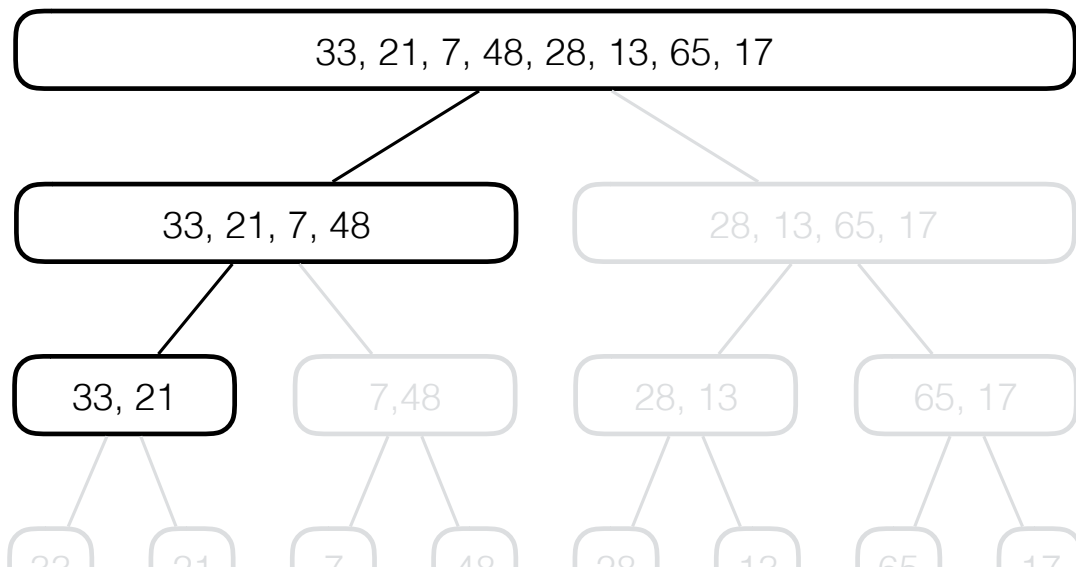
28

13

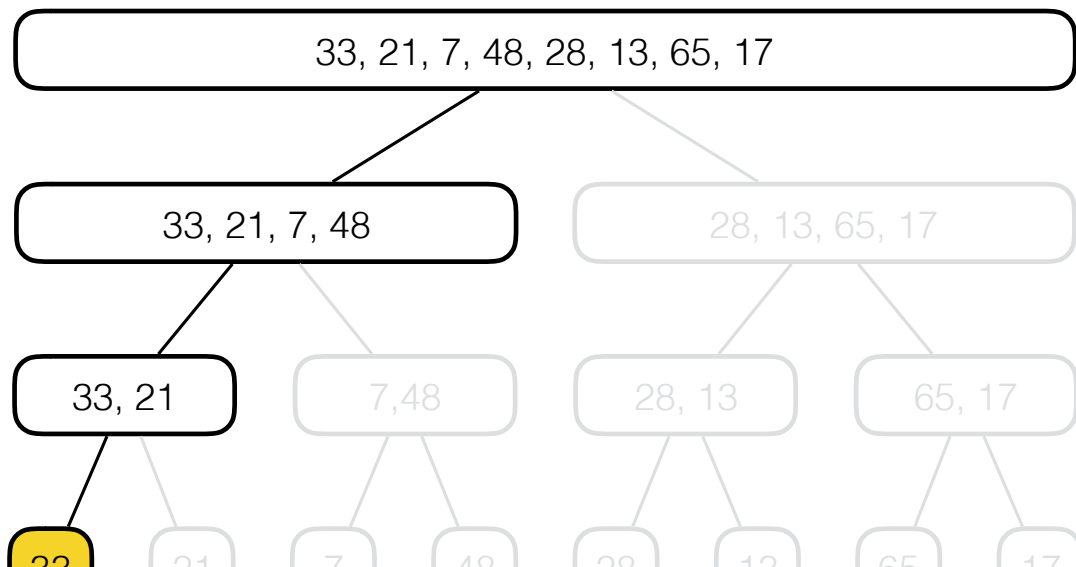
65

17

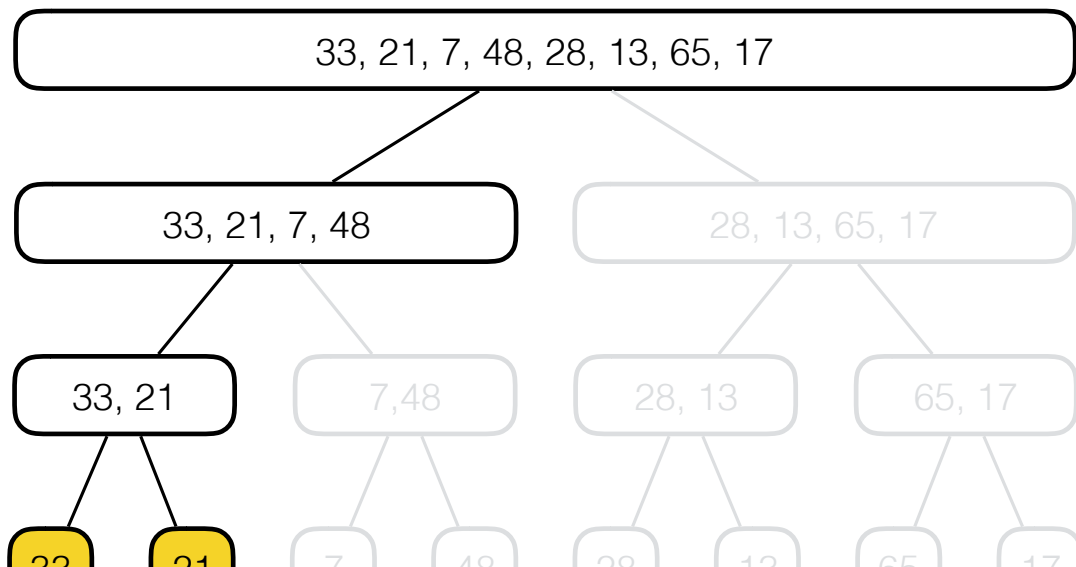
MergeSort(): Esecuzione



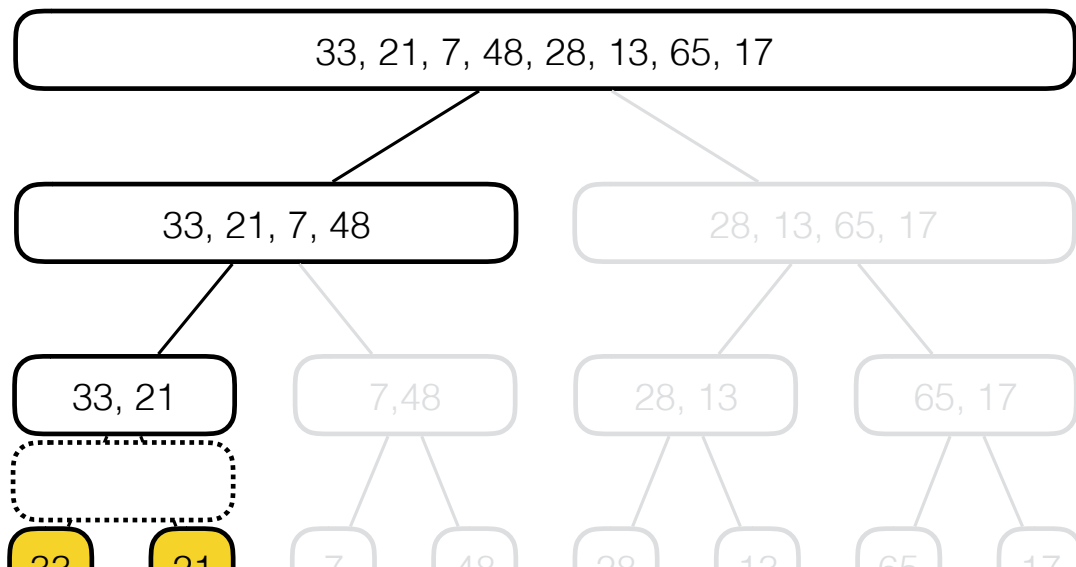
MergeSort(): Esecuzione



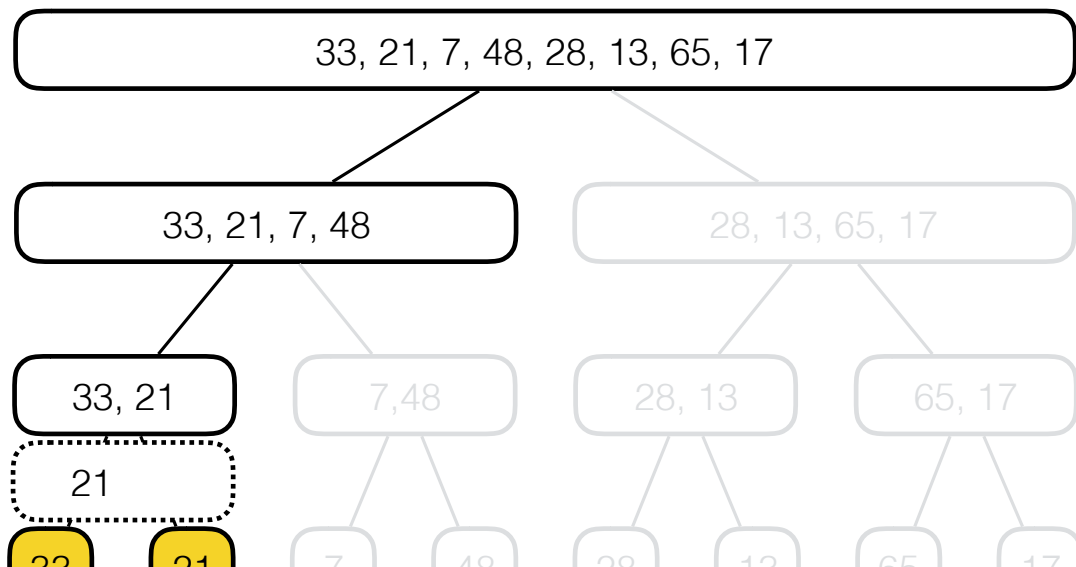
MergeSort(): Esecuzione



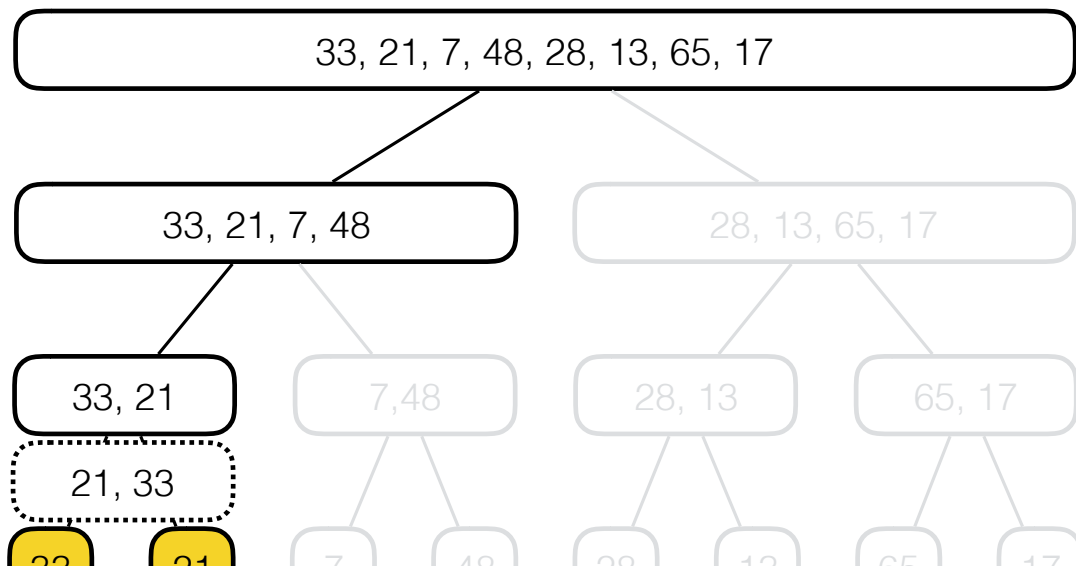
MergeSort(): Esecuzione



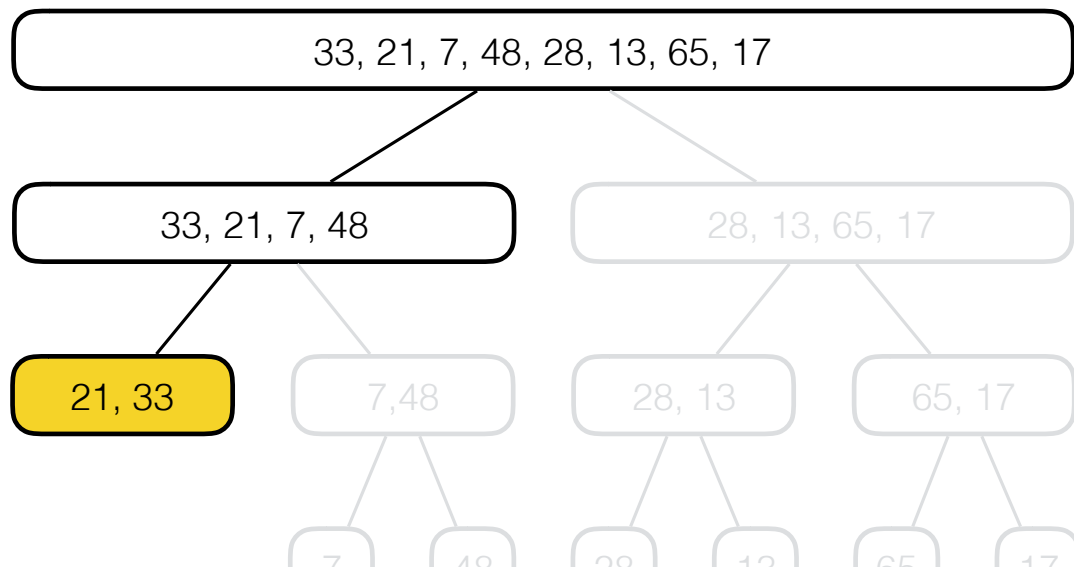
MergeSort(): Esecuzione



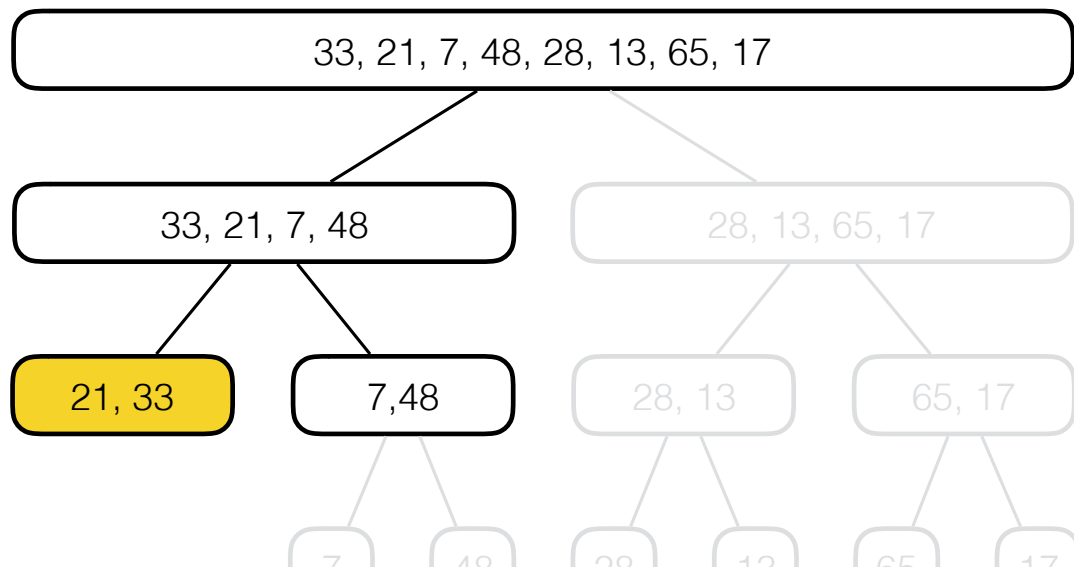
MergeSort(): Esecuzione



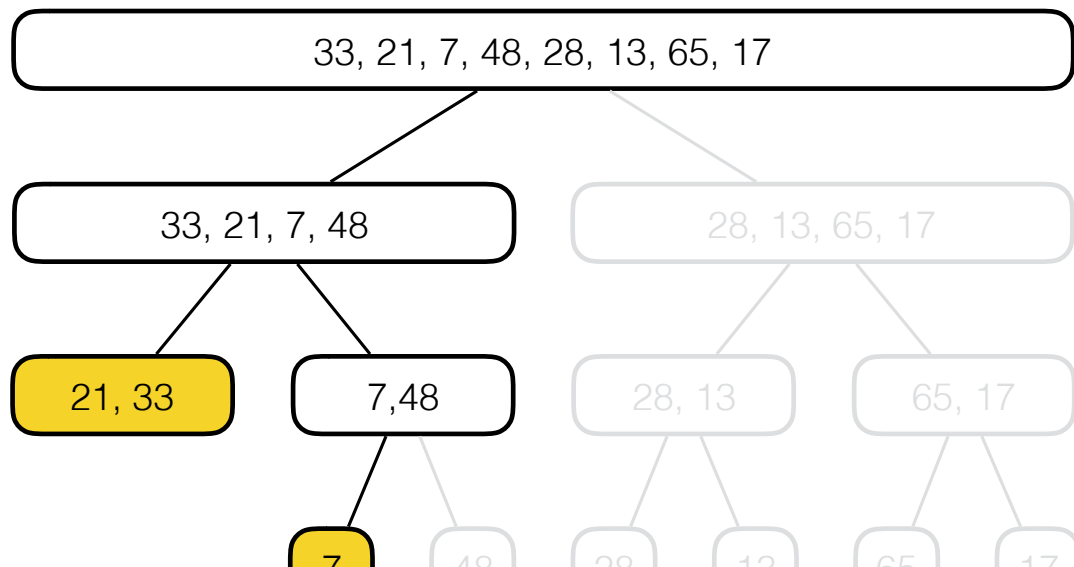
MergeSort(): Esecuzione



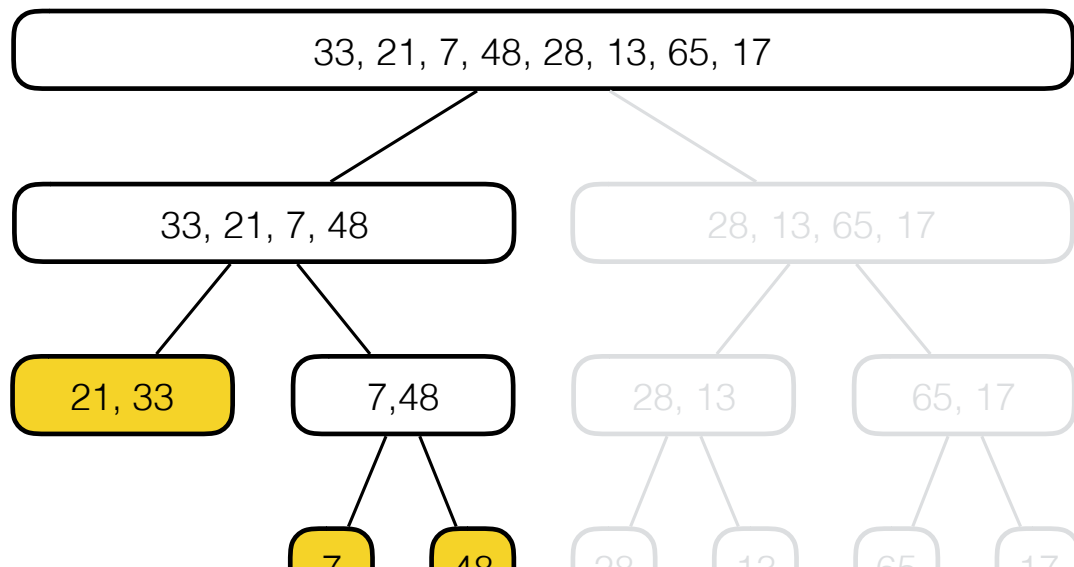
MergeSort(): Esecuzione



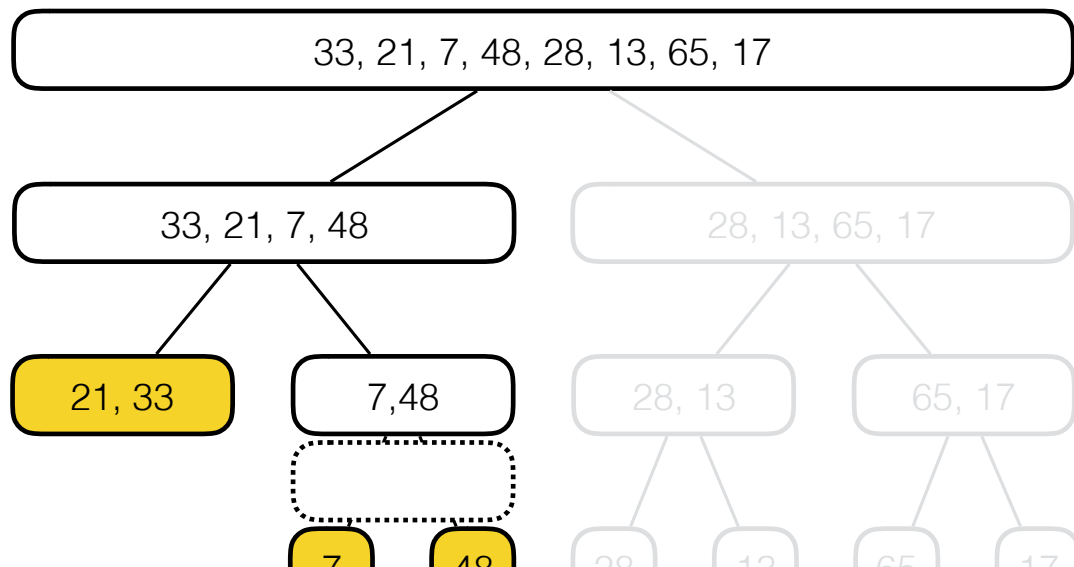
MergeSort(): Esecuzione



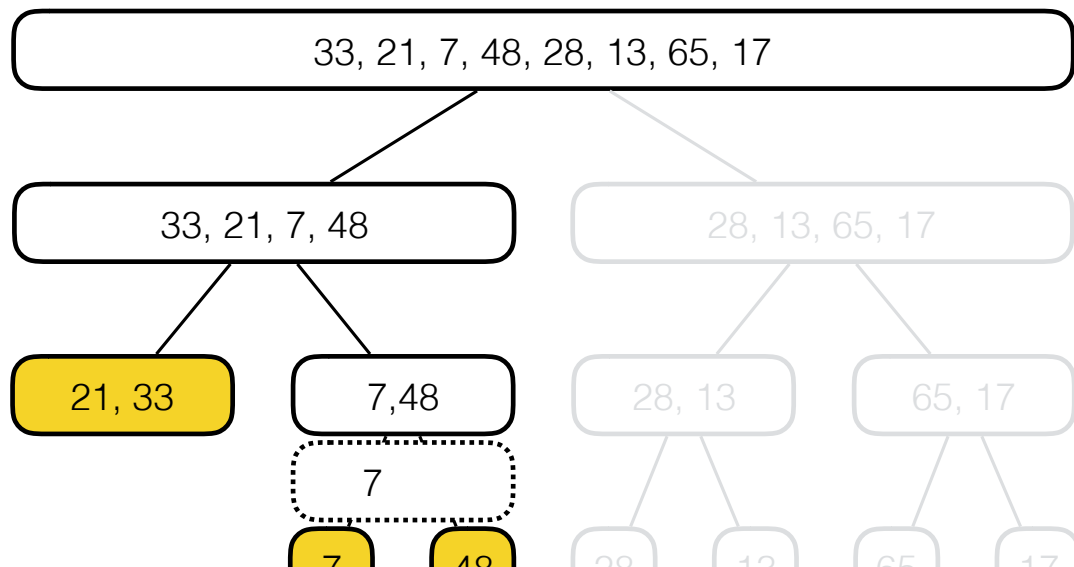
MergeSort(): Esecuzione



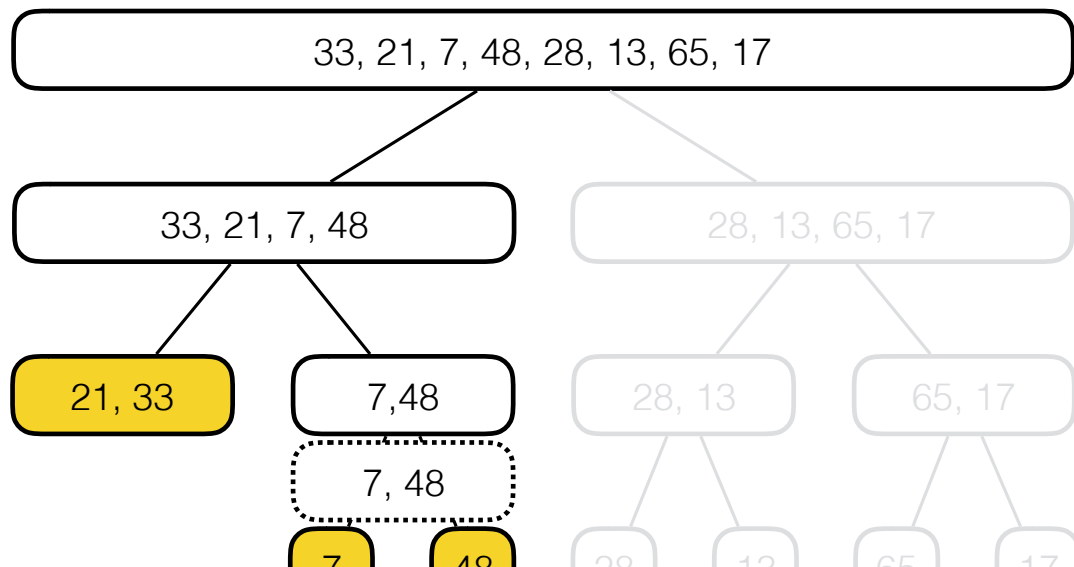
MergeSort(): Esecuzione



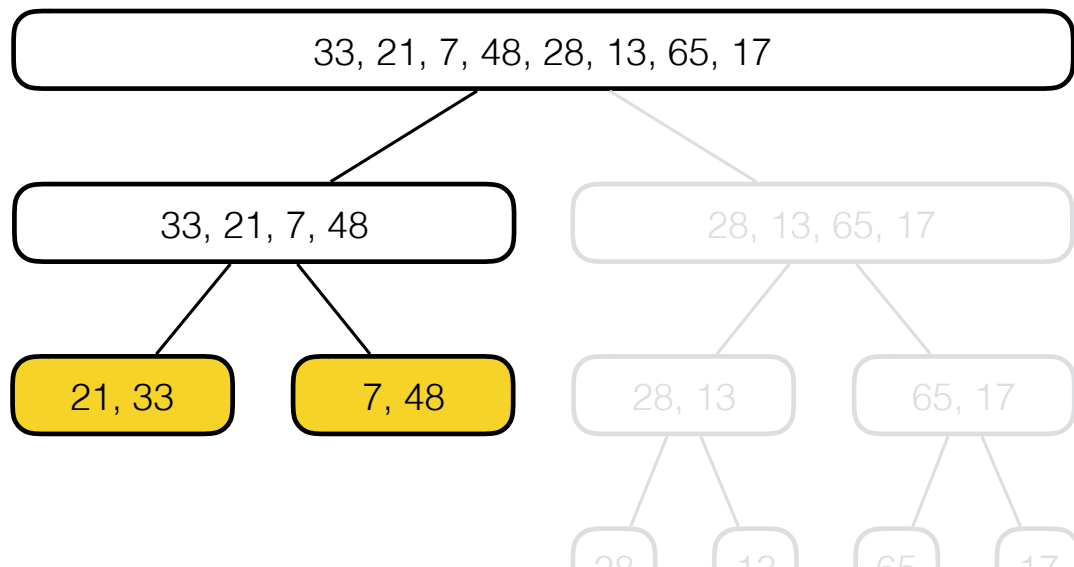
MergeSort(): Esecuzione



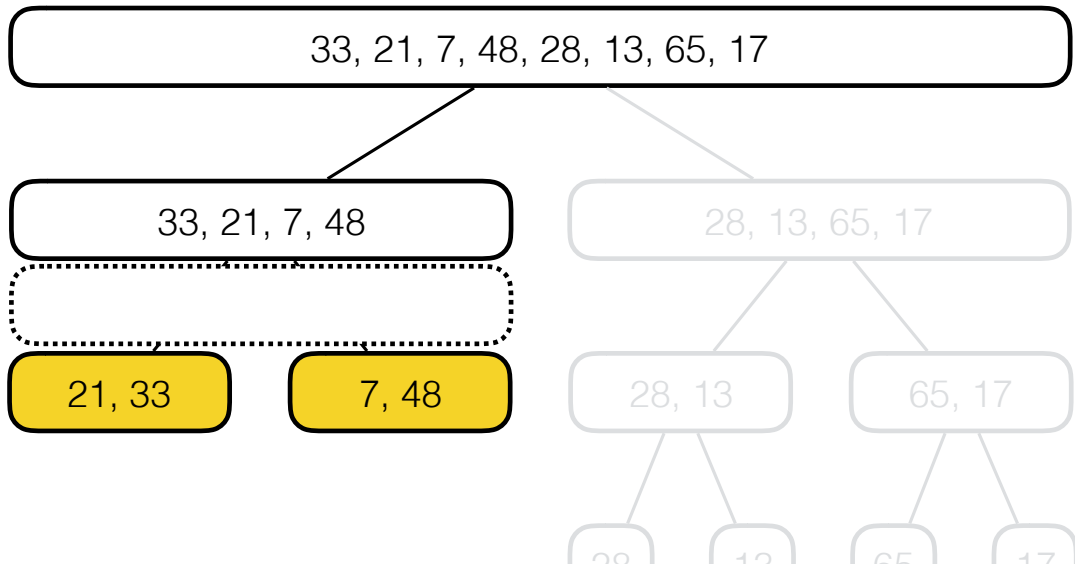
MergeSort(): Esecuzione



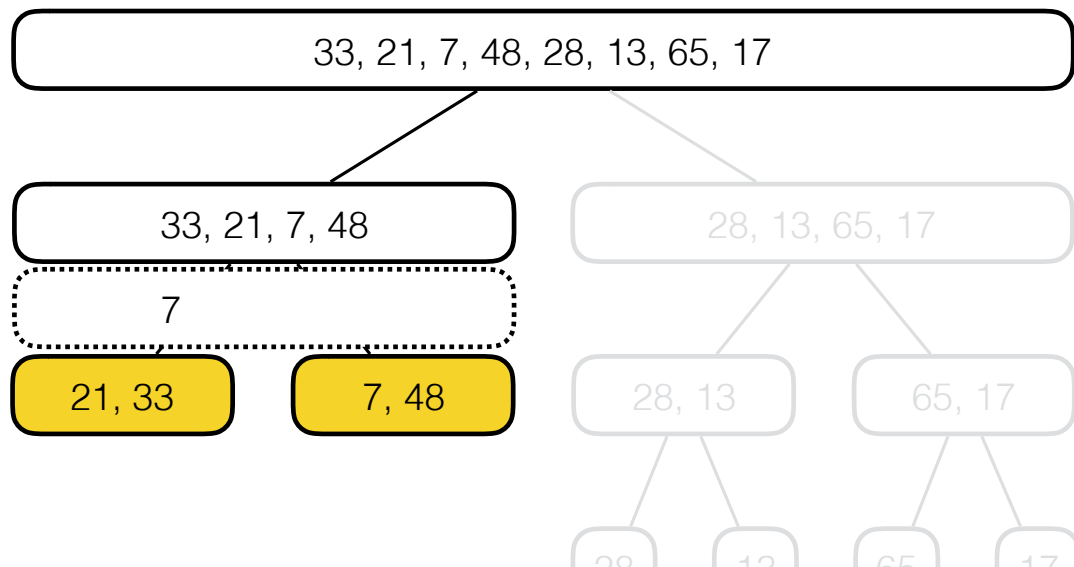
MergeSort(): Esecuzione



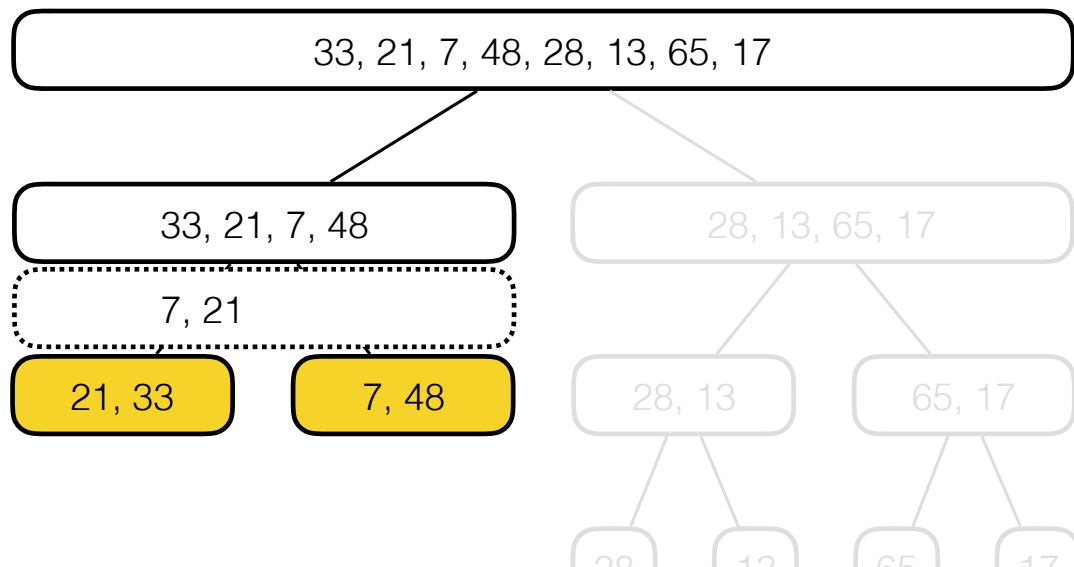
MergeSort(): Esecuzione



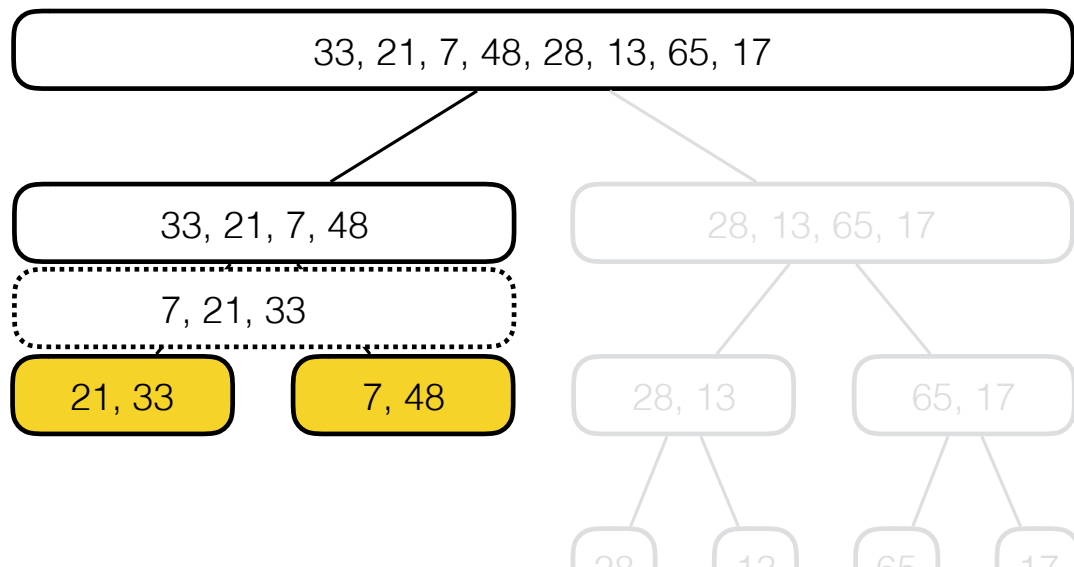
MergeSort(): Esecuzione



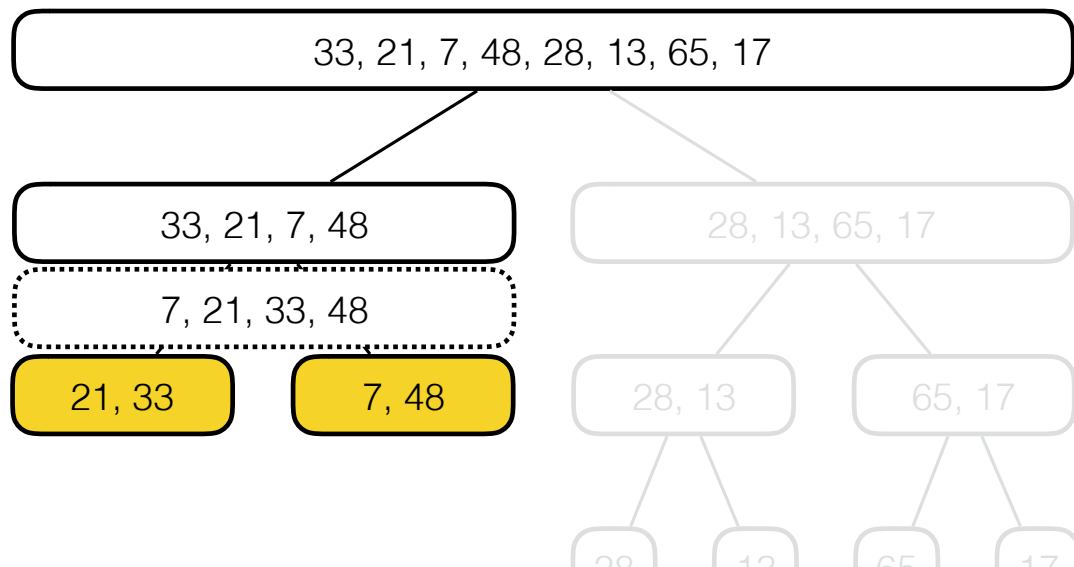
MergeSort(): Esecuzione



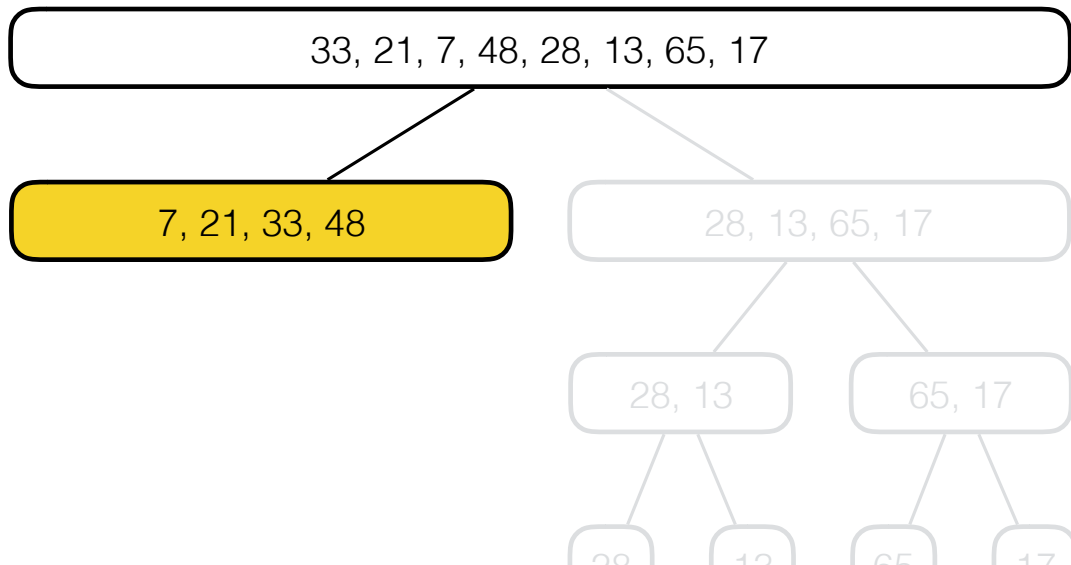
MergeSort(): Esecuzione



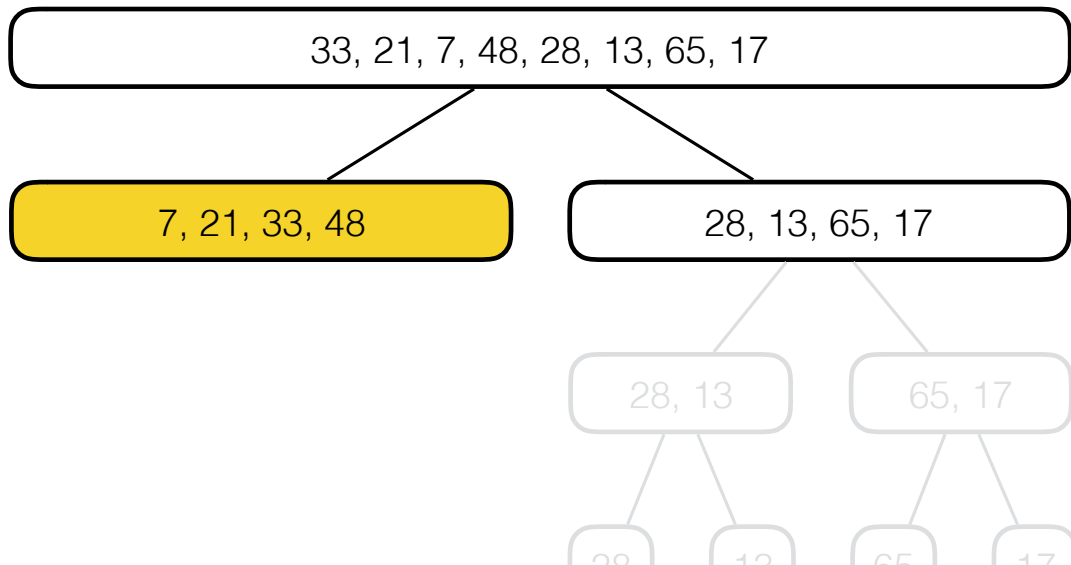
MergeSort(): Esecuzione



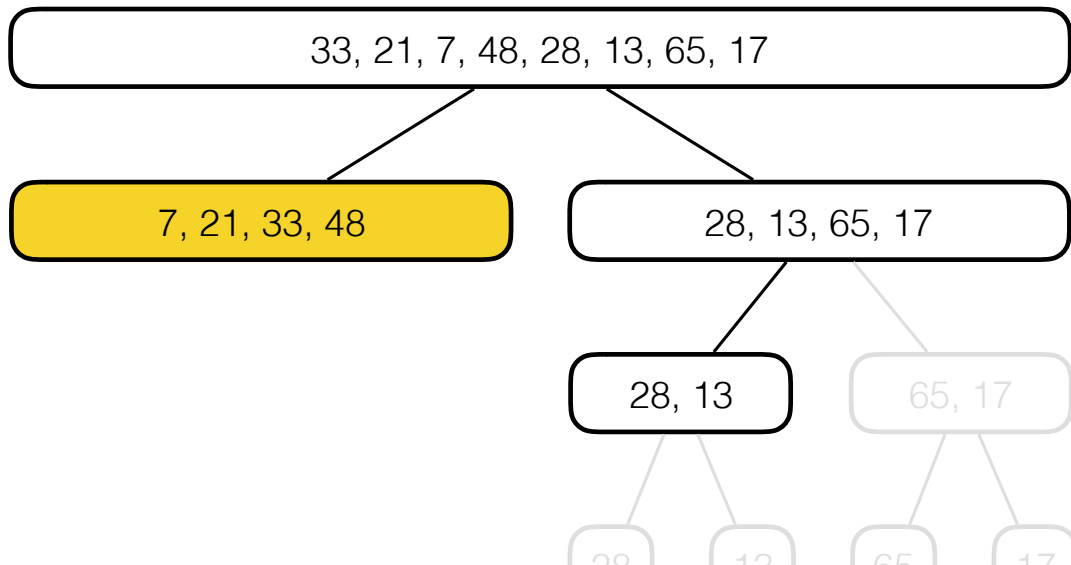
MergeSort(): Esecuzione



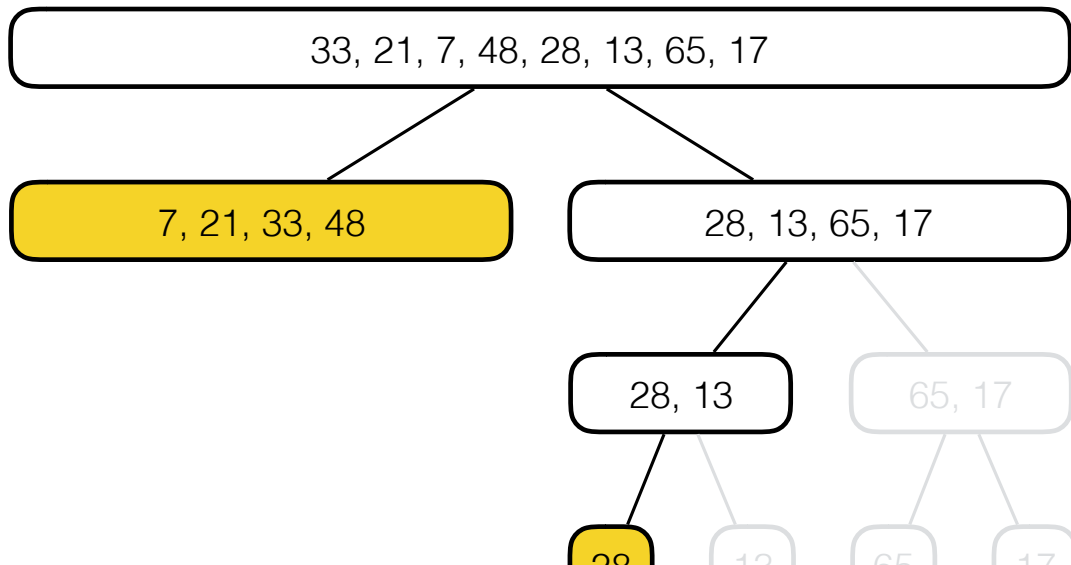
MergeSort(): Esecuzione



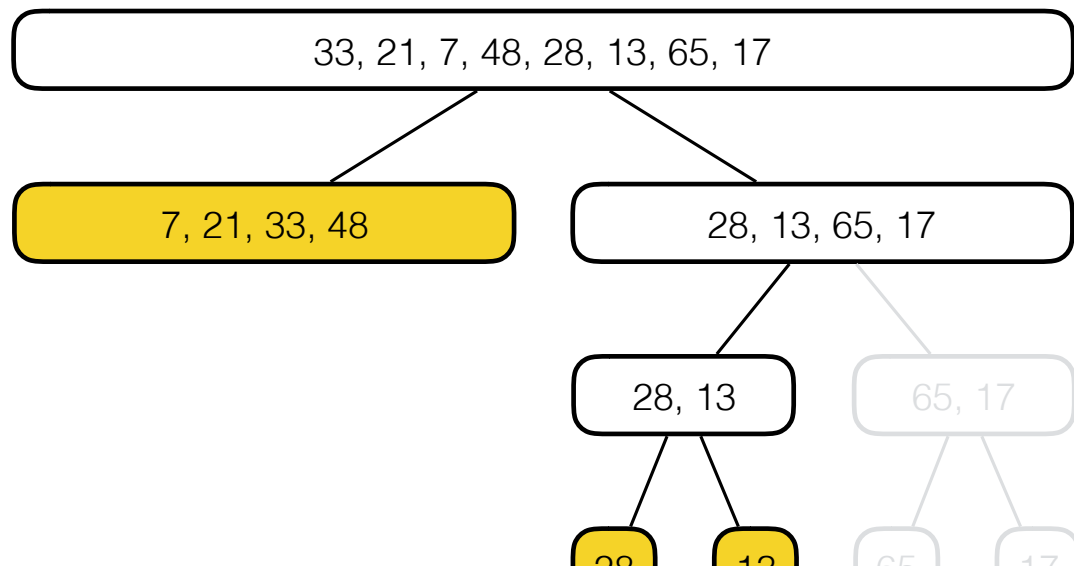
MergeSort(): Esecuzione



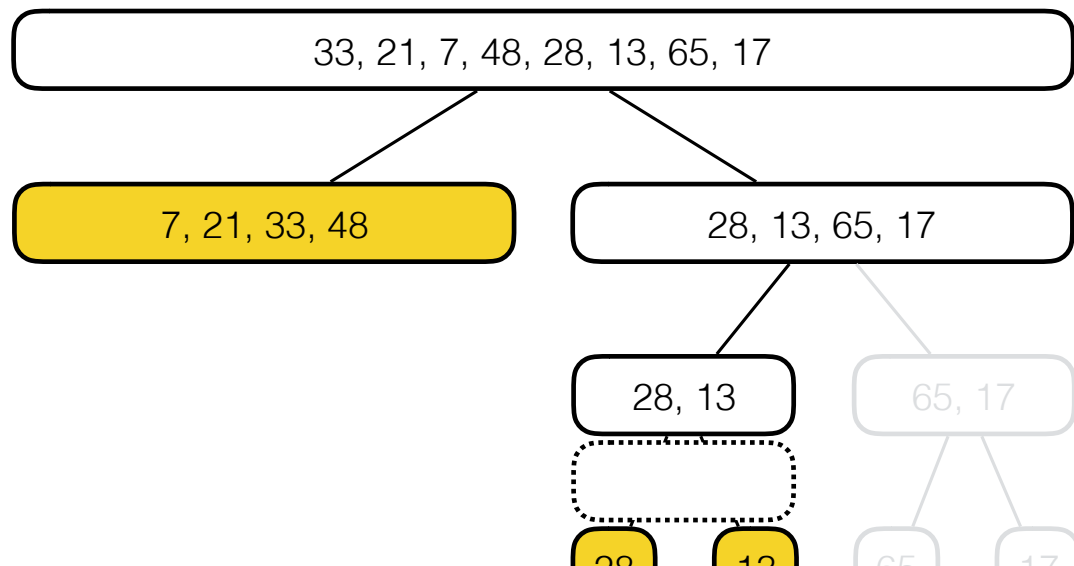
MergeSort(): Esecuzione



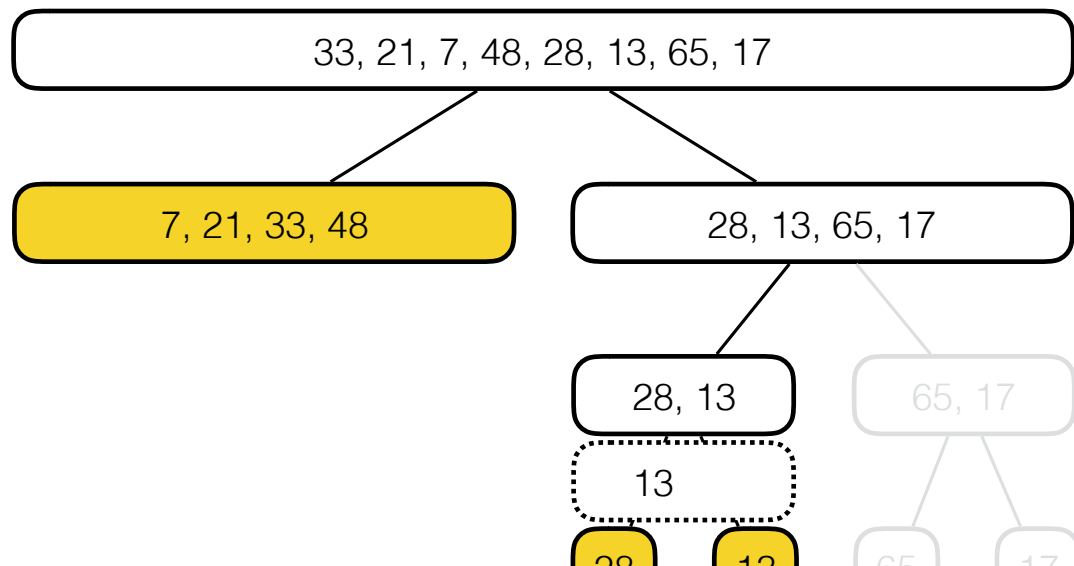
MergeSort(): Esecuzione



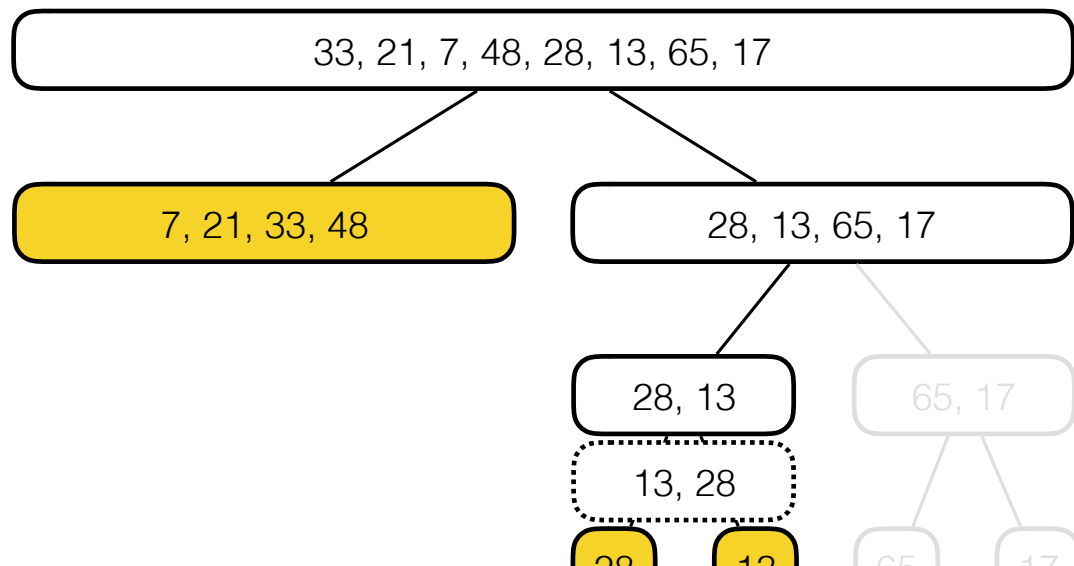
MergeSort(): Esecuzione



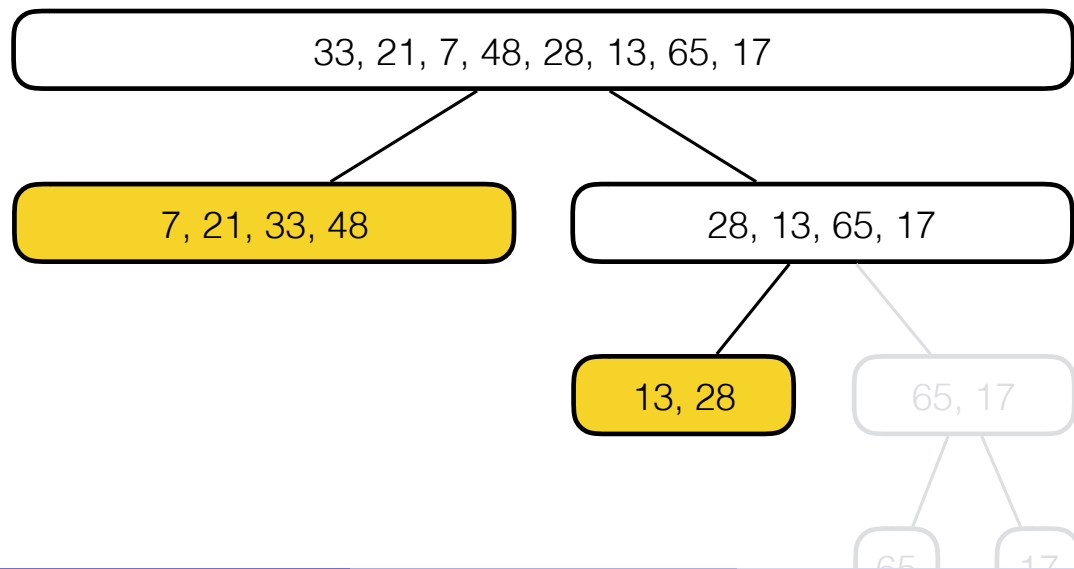
MergeSort(): Esecuzione



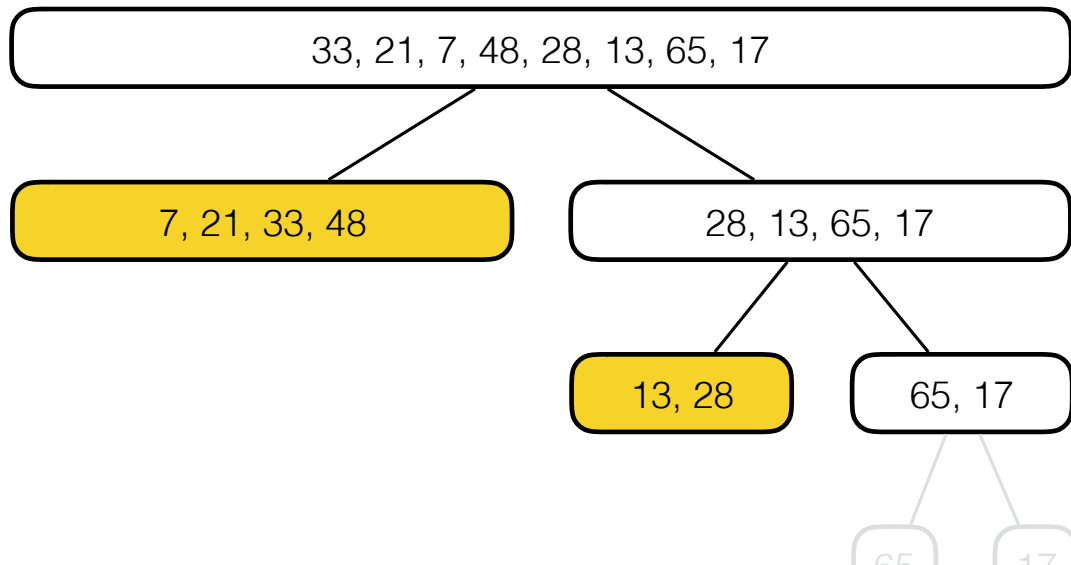
MergeSort(): Esecuzione



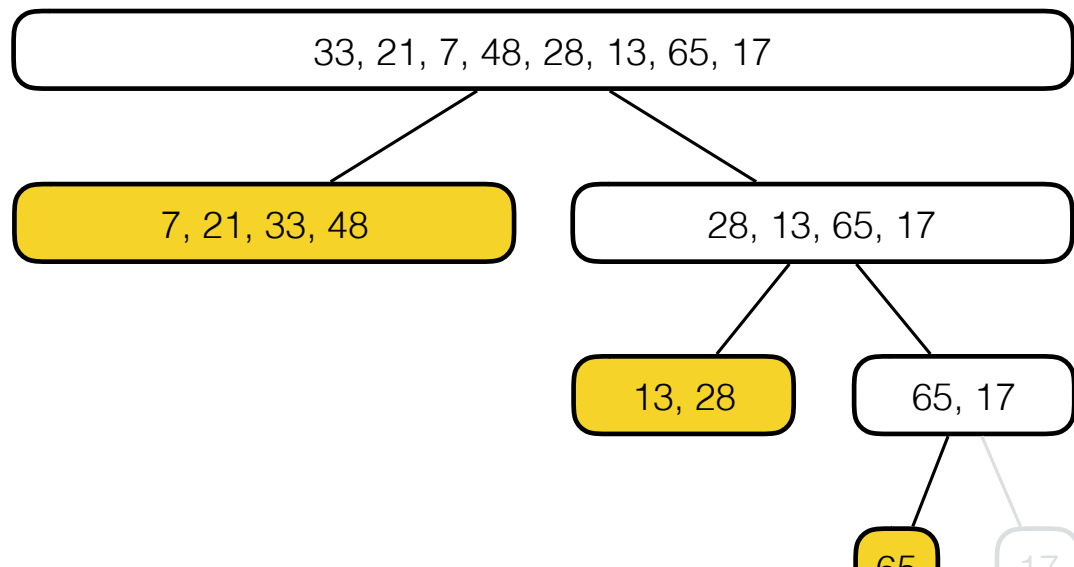
MergeSort(): Esecuzione



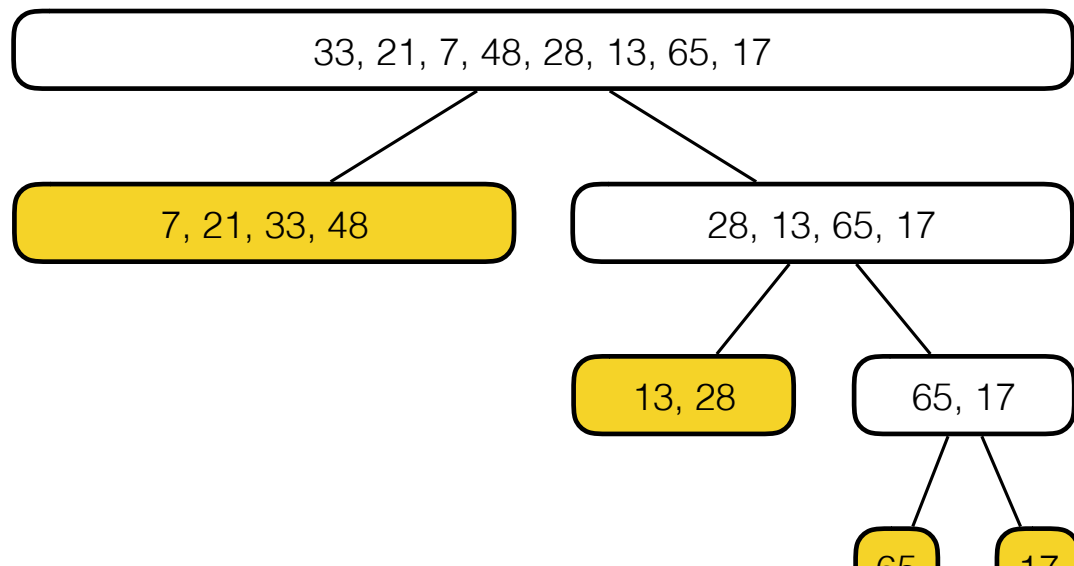
MergeSort(): Esecuzione



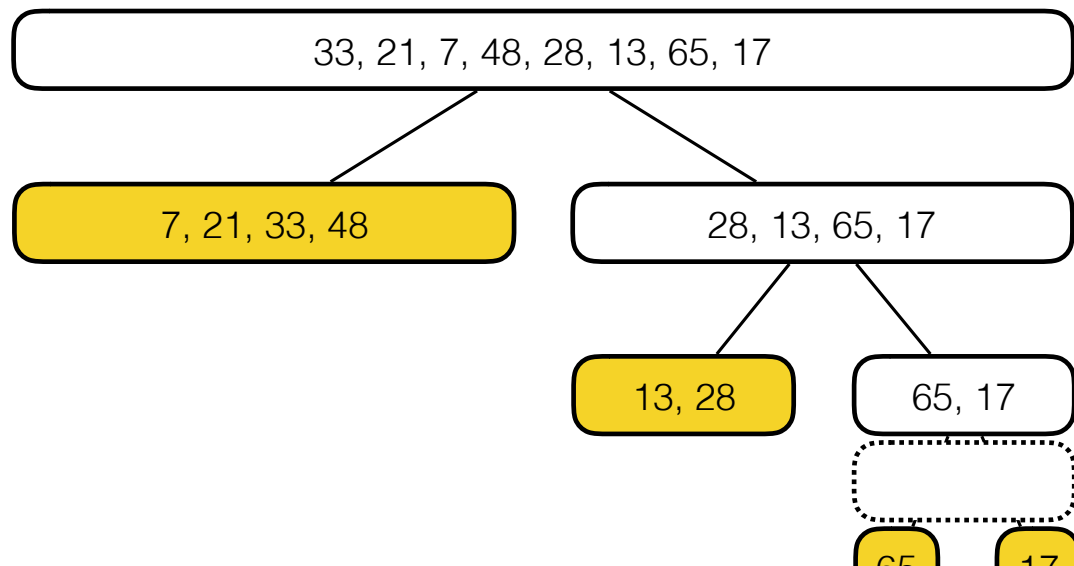
MergeSort(): Esecuzione



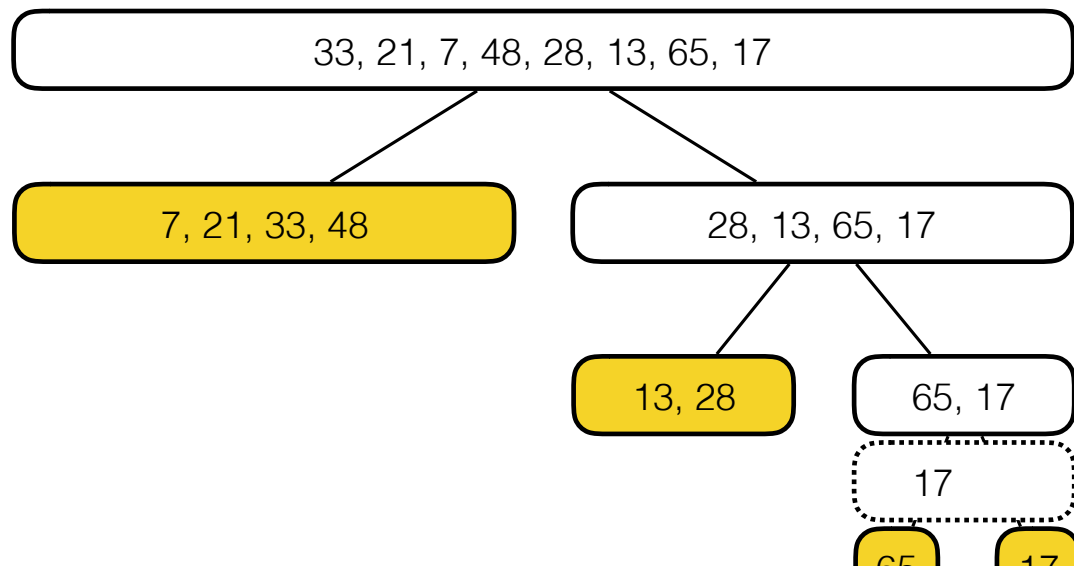
MergeSort(): Esecuzione



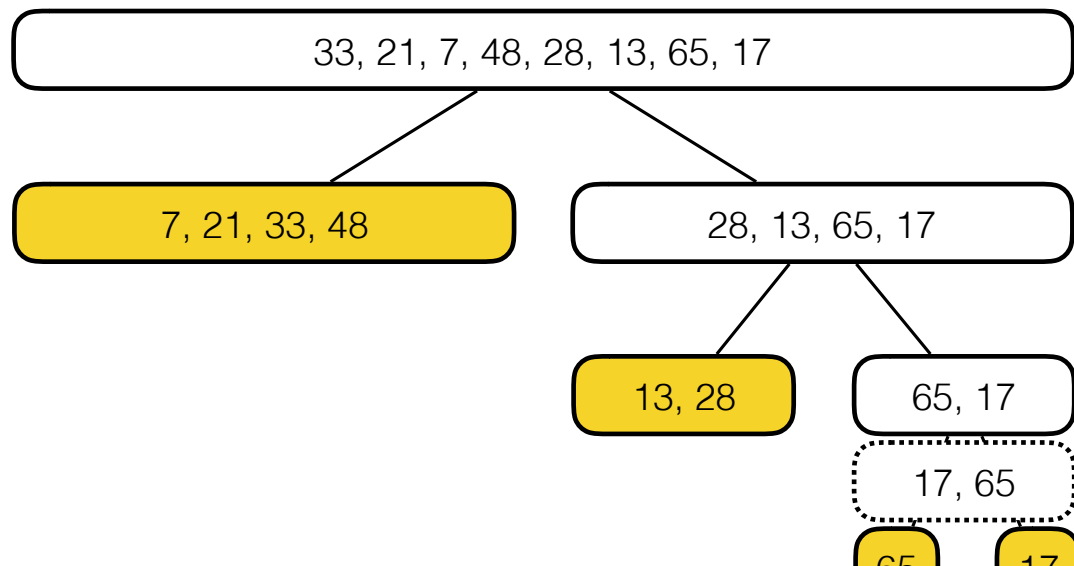
MergeSort(): Esecuzione



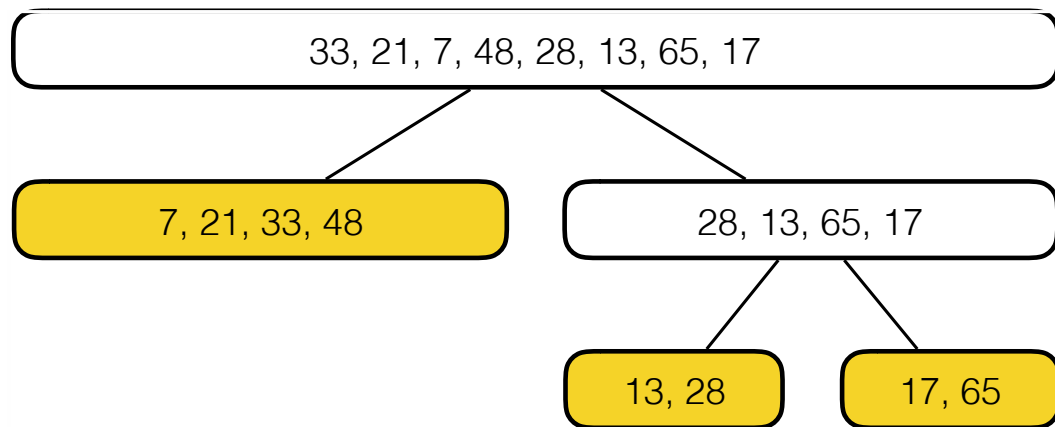
MergeSort(): Esecuzione



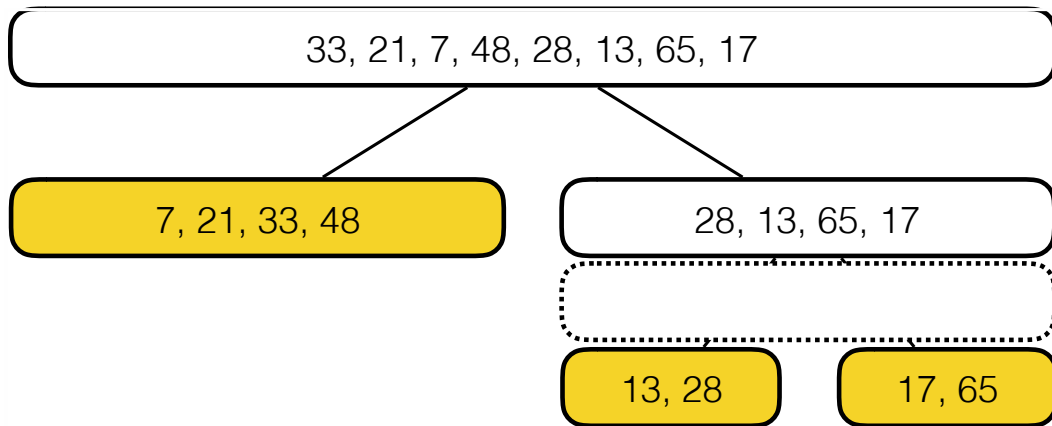
MergeSort(): Esecuzione



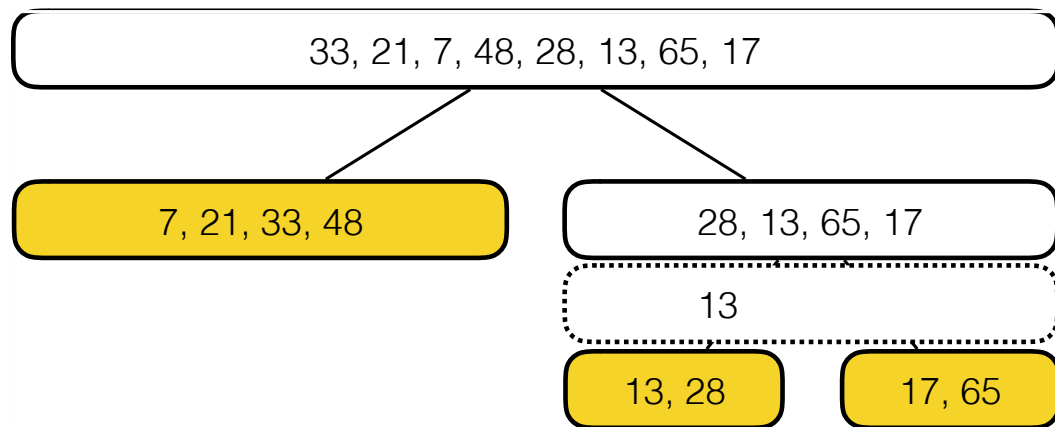
MergeSort(): Esecuzione



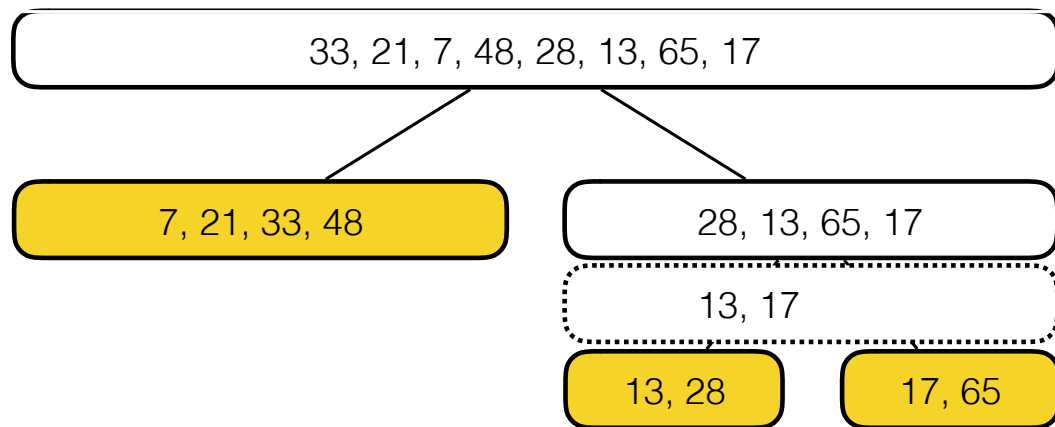
MergeSort(): Esecuzione



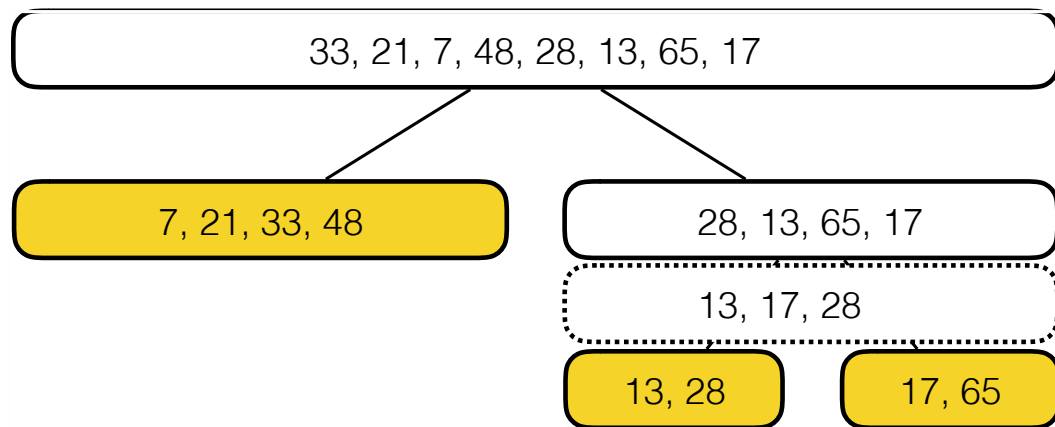
MergeSort(): Esecuzione



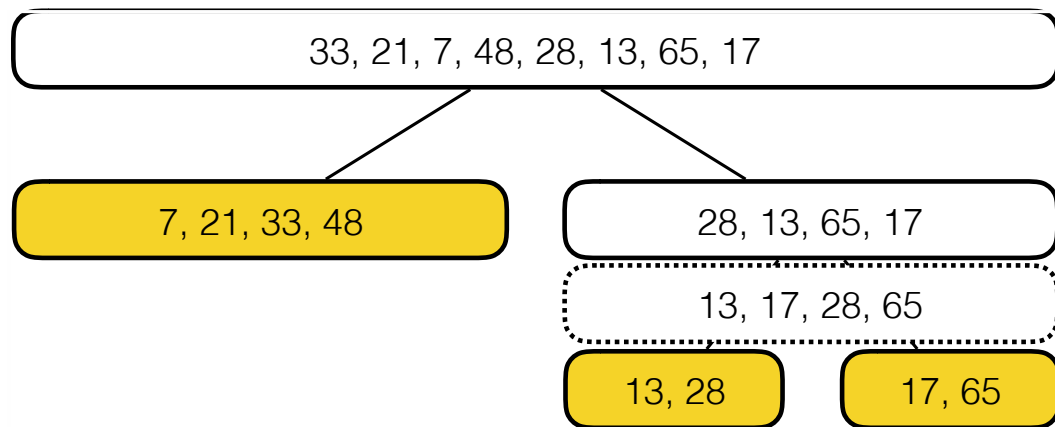
MergeSort(): Esecuzione



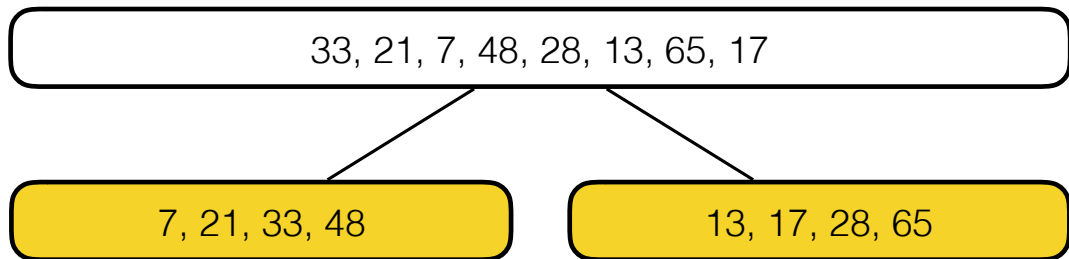
MergeSort(): Esecuzione



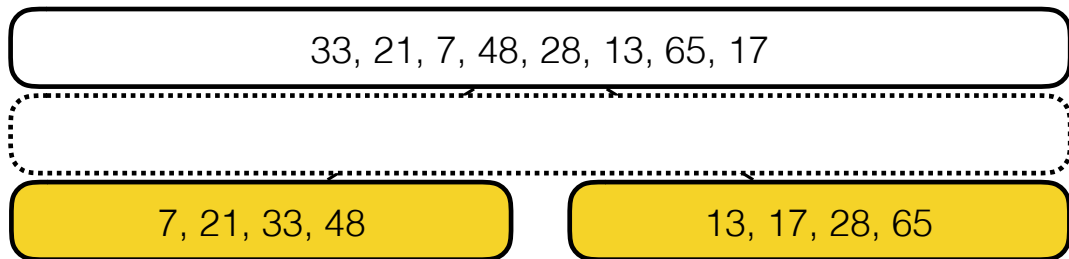
MergeSort(): Esecuzione



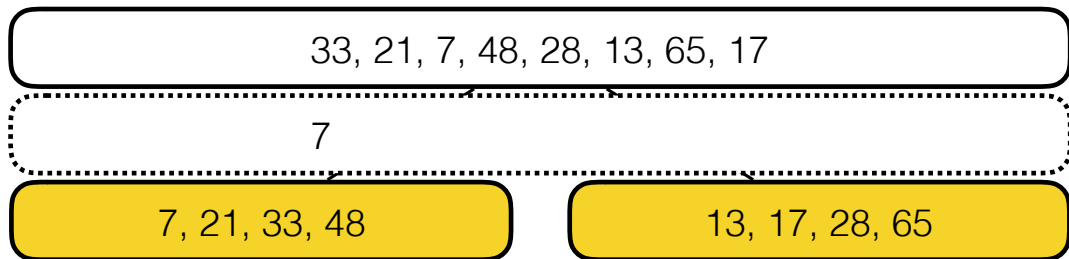
MergeSort(): Esecuzione



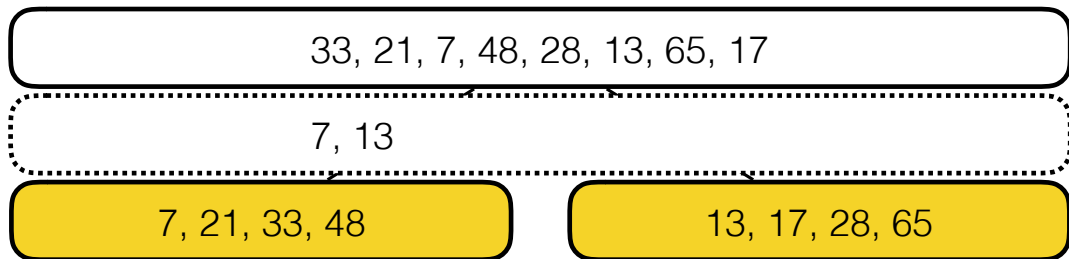
MergeSort(): Esecuzione



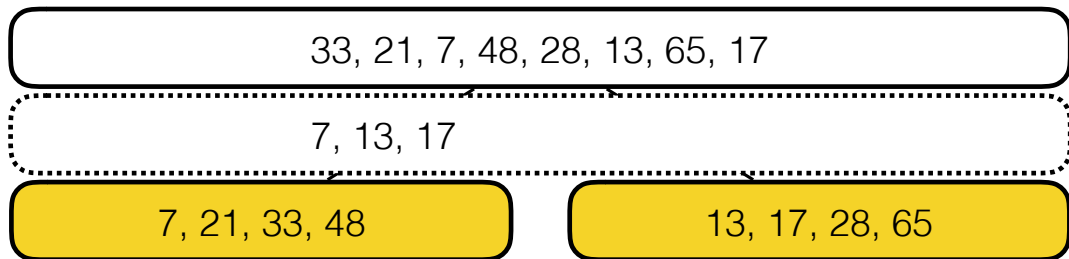
MergeSort(): Esecuzione



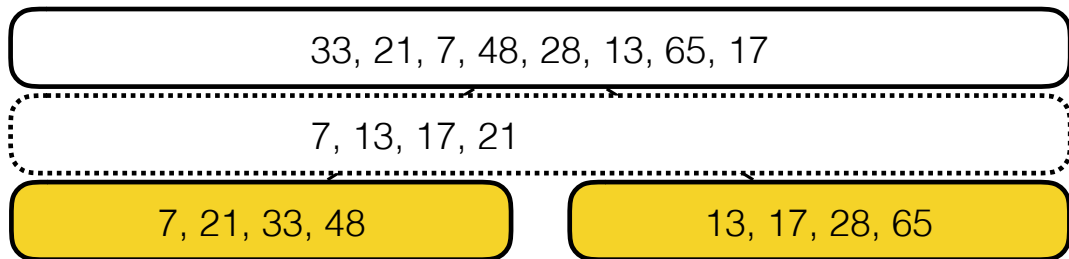
MergeSort(): Esecuzione



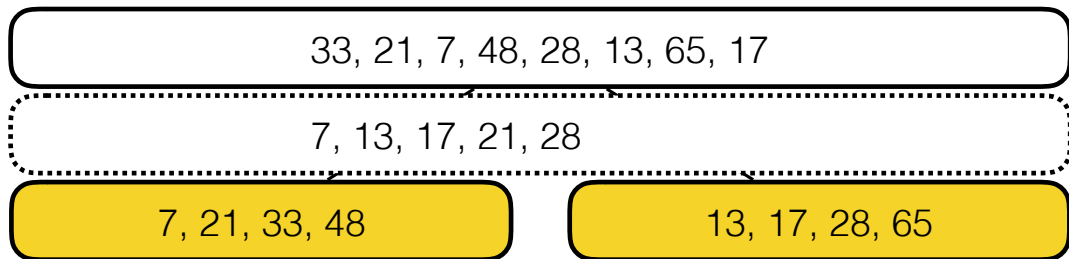
MergeSort(): Esecuzione



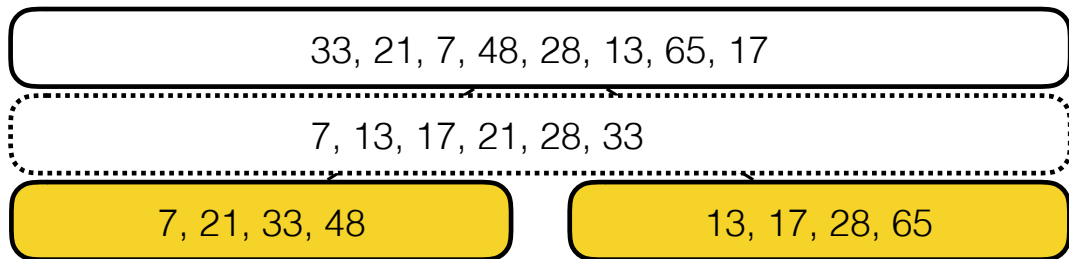
MergeSort(): Esecuzione



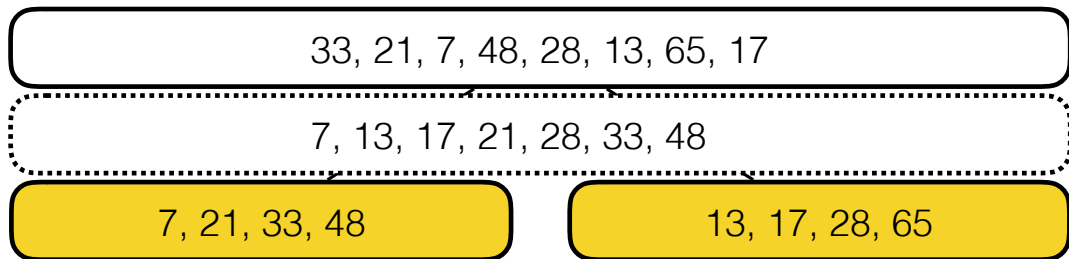
MergeSort(): Esecuzione



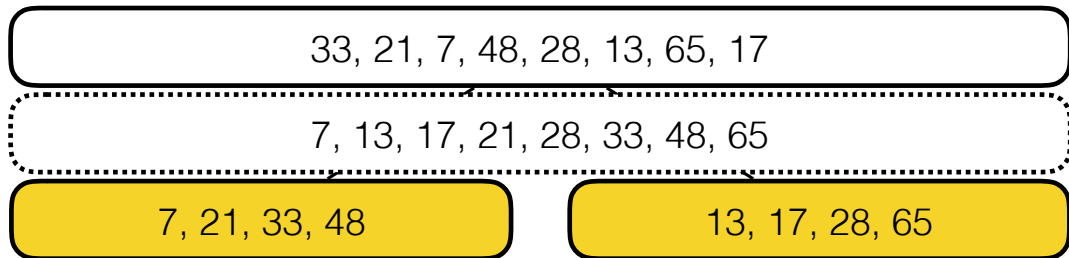
MergeSort(): Esecuzione



MergeSort(): Esecuzione



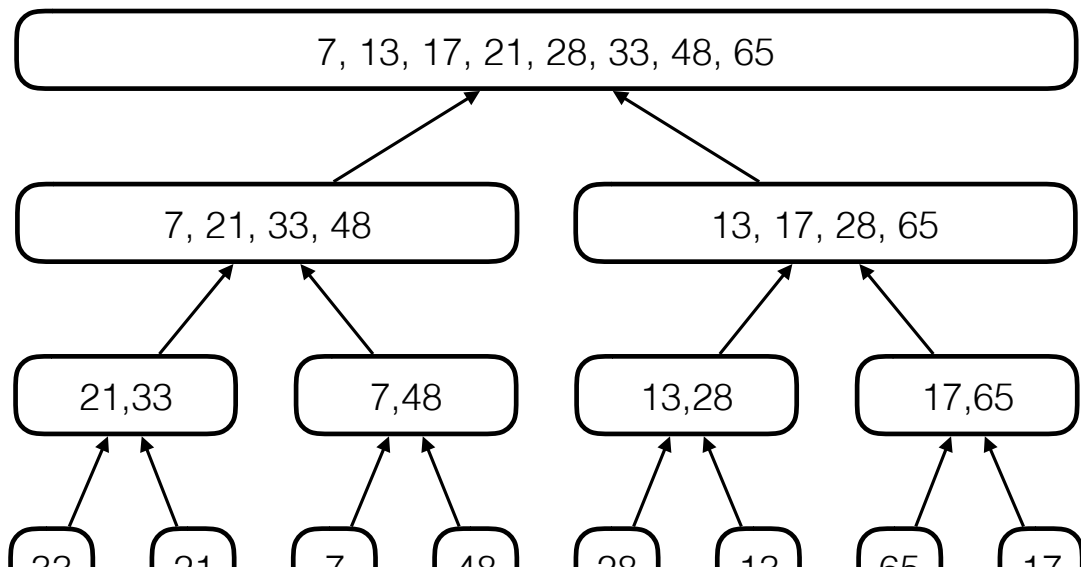
MergeSort(): Esecuzione



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

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

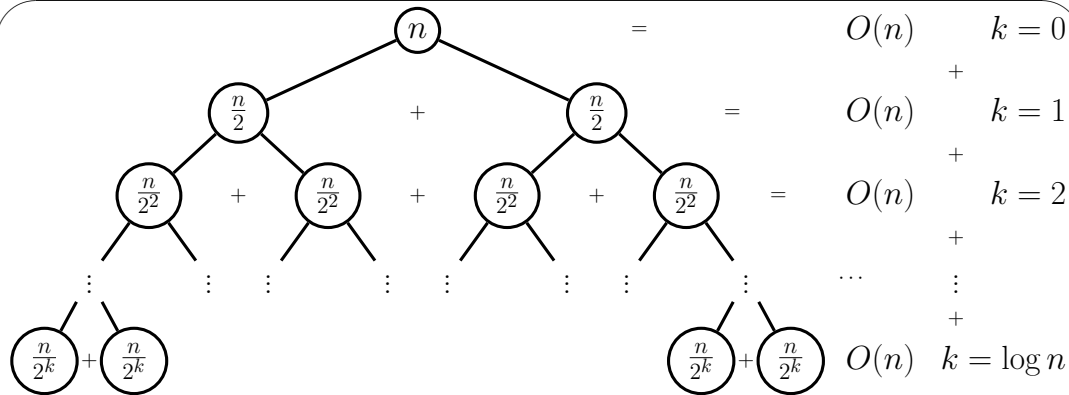
Costo computazionale di Merge Sort

Domanda

Qual è il costo computazionale di MergeSort()?

Domanda

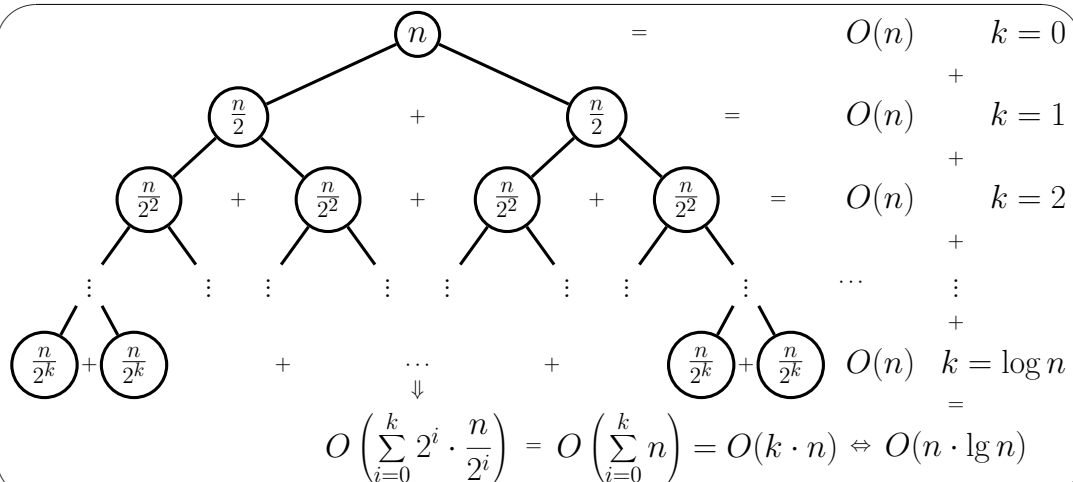
Qual è il costo computazionale di MergeSort()?



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.



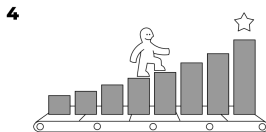
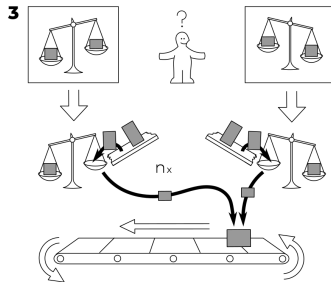
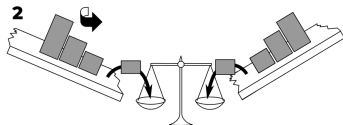
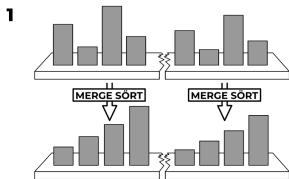
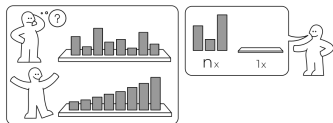
Hollerith Machine

MergeSort, in pillole

MERGE SÖRT

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

IDEA



Ricorsione

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

Alan Perlis, Epigrams on Programming

The image is a screenshot of a Google search interface. At the top, there is a navigation bar with links for '+Vishal', 'Web', 'Images', 'Maps', 'News', 'Orkut', 'Gmail', and 'More'. Below this is the Google logo on the left and a search input field containing the text 'recursion'. Under the search bar, it says 'Search' and 'About 1,570,000 results (0.20 seconds)'. On the left side, there is a vertical menu with links for 'Everything', 'Images', 'Maps', 'Videos', 'News', and 'More'. Below this menu, it says 'Pune, Maharashtra' and 'Change location'. On the right side, there is a suggestion box that says 'Did you mean: [recursion](#)'. Below this, there are two search results. The first result is titled 'Recursion - Wikipedia, the free encyclopedia' and includes the URL 'en.wikipedia.org/wiki/Recursion'. The second result is titled 'Recursion (computer science) - Wikipedia, the free encyclopedia' and includes the URL 'en.wikipedia.org/wiki/Recursion_(computer_science)'. Both results include a brief description of recursion.

+Vishal Web Images Maps News Orkut Gmail More ▾

Google recursion

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

Everything
Images
Maps
Videos
News
More

Pune, Maharashtra
Change location

Did you mean: [recursion](#)

[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 - Recursi

[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