1 Analisi della complessità degli algoritmi

Obbiettivo

L'obiettivo dell'**analisi** degli algoritmi è quello di **stimare la loro complessità** in termini di **tempo di calcolo**.

Dunque, l'analisi della complessità degli algoritmi torna **utile** per **svariati motivi**:
- Stimare il tempo impiegato per un dato in input;
- Stimare il più grande input gestibile in tempi ragionevoli;
- Confrontare l'efficienza di algoritmi diversi;
- Ottimizzare le parti più importanti.

1.1 Complessità e dimensione dell'input

Cos'è la complessità?

Possiamo definire la **complessità** come una funzione matematica che descrive l'andamento del tempo di calcolo in relazione alla ***dimensione dell'input***.
$T : "Dimensione\ dell'input" \rightarrow "Tempo\ di\ calcolo"$

Dunque, maggiore è la dimensione dell'input, maggiore è il tempo di calcolo, con una **conseguente crescita** delle **risorse impiegate** dall'algoritmo per risolvere il problema.
Le due principali tipologie di risorse impiegate sono:
- *Risorse di **complessità temporale***: cioè la quantità di tempo richiesta dall'algoritmo;
- *Risorse di **complessità spaziale***: la quantità di memoria necessaria durante l'esecuzione.

In realtà, *complessità spaziale* diventa un problema secondario, che si andrà ad analizzare solo nel caso in cui, confrontando due algoritmi per determinarne il "*migliore*", abbiano la stessa complessità in termini di tempo.

Dimensione dell'input

Con dimensione dell'input intendiamo la sua **taglia**, e abbiamo due possibili casi:

- *Criterio di **costo uniforme***: la taglia dell'input è il **numero di elementi di cui è costituito**. In altre parole, ogni elemento dell'input costa 1 unità, **indipendentemente** da quanti **bit servono per rappresentarlo**.
  Esempio: per la ricerca del minimo in un vettore di n elementi, l'algoritmo che lo elabora avrà un costo **proporzionale a** n.
- *Criterio di **costo logaritmico***: la taglia dell'input è il **numero di bit necessari per rappresentarlo**. In questo caso, il costo dell'elaborazione dipende direttamente dalla **lunghezza in bit** dei dati in ingresso, e non dal numero di elementi di cui è composto.
  Esempio: avendo in ingresso un numero intero molto grande, si può considerare il numero di bit necessari per rappresentarli.

Nella pratica, se ogni elemento dell'input occupa un numero costante di bit, allora i due criteri (uniforme e logaritmico) forniscono risultati equivalenti, **a meno** di una **costante moltiplicativa** applicata alla dimensione dell'**input**. Ad esempio, un input costituito da n byte (criterio di costo uniforme) corrisponde a $8n$ bit (criterio di costo logaritmico).

1.2 Definizione di tempo e modello di calcolo

Tuttavia, come introdotto al capitolo **??**, l'approccio più immediato per valutare il **tempo di esecuzione/calcolo** di un algoritmo, non è quello di misurare i secondi impiegati dal calcolatore poiché entrerebbero in gioco dei fattori esterni, non dipendenti dall'algoritmo stesso. Quindi misurando solo "*quanti secondi impiega un'algoritmo*", non si possono confrontare gli algoritmi in modo universale, ma solo "*sul computer in quel determinato momento*".

L'**analisi della complessità** vuole invece essere **indipendente dal calcolatore**, proprio per questo, un **approccio** sicuramente **migliore** è quello di considerare come "*tempo di calcolo*" il numero di istruzioni elementari eseguite.

Tempo ≡ numero istruzioni elementari

Un'**istruzione** viene considerata **elementare** se può essere eseguita in tempo "*costante*" dal processore (Il tempo di esecuzione ***corrisponde*** al numero di istruzioni elementari).

Cos'è il tempo di calcolo?

Poiché i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati di ingresso, viene spontaneo esprimere il **tempo di calcolo** come: *il* ***costo complessivo*** *delle* ***operazioni elementari*** *in funzione della* ***dimensione*** n ***dei dati*** *in ingresso*.

Per poter capire quali istruzioni debbano essere considerate elementari, si utilizza il concetto di **modello di calcolo**, ovvero una rappresentazione semplificata ma rigorosa di un calcolatore, che definisce **quali operazioni** sono **ammesse** e **quanto costano**, tutto ciò in modo **indipendente** dalle caratteristiche **dell'hardware**.
Un buon modello di calcolo **soddisfa tre requisiti** fondamentali:

- ***Astrazione***: deve permettere di **ignorare** i **dettagli irrilevanti** del calcolatore (ad esempio, non interessa conoscere la tipologia di processore e di quanta memoria disponga);
- ***Realismo***: deve riflettere una situazione reale;
- ***Potenza matematica***: deve consentire di trarre conclusioni matematiche (formali) sul costo computazionale.

1.2.1 Random Access Machine (RAM)

Il modello di calcolo che normalmente viene utilizzato è detto **Random Access Machine (RAM)** ed è caratterizzato da:

- ***Memoria***: si assume che la memoria presenti una quantità infinita di celle di dimensione finita, poiché si vuole capire il funzionamento dell'algoritmo al crescere della dimensione dell'input, ciascuna delle quali è accessibile in tempo costante;
- ***Processore (singolo)***: ha un processore singolo che esegue un insieme limitato di **istruzioni elementari** (come somma, sottrazione, moltiplicazione, operazioni logiche, salti condizionati, ecc. . .);
- ***Costo delle istruzioni elementari***: ad ogni istruzione elementare viene assegnato, un **costo costante**, che rappresenta il tempo richiesto per eseguirla.

Lo scopo finale non è confrontare le prestazioni dei processori (compito dei benchmark), ma determinare se un algoritmo è più efficiente di un altro.

1.3 Valutazione del caso pessimo, medio e ottimo

Principalmente, il costo delle singole operazioni è valutato nel **caso pessimo**, ovvero sul dato di ingresso più sfavorevole tra tutti quelli di dimensione n.
In alternativa, si può considerare anche il **caso medio**, calcolando la media dei costi su tutti i possibili input di dimensione n, pesata in base alla probabilità con cui ciascun dato può verificarsi. Mentre, talvolta si introduce anche il **caso ottimo** che rappresenta il costo minimo dell'algoritmo su un input di diemnsione n.

Il caso ottimo è **raramente utile**: infatti un buon comportamento nel caso ottimo **non garantisce prestazioni accettabili negli altri casi** e può dare un'**illusione di efficienza** non rappresentativa del comportamento complessivo dell'algoritmo.

Perché valutiamo il caso pessimo se può verificarsi molto raramente?

Il vantaggio del caso pessimo è dato dal fatto che questo tipo di valutazione non richiederà mai, per nessun dato di dimensione n, un tempo maggiore.
Invece, la valutazione nel caso medio sembra più realistica, ma è ignota la distribuzione di probabilità: spesso viene utilizzata una distribuzione uniforme che per molti problemi è irrealistica

1.3.1 Tempo di calcolo della funzione `min()` (iterativa)

In questo caso si vuole stimare il tempo di calcolo della funzione `min()` che si occupa di trovare l'elemento più piccolo all'interno di un vettore. Per prima cosa si controlla il numero delle operazioni elementari dalla quale è costituita la funzione in esame.
A questo punto possiamo:

- Indicare con C_h il costo richiesto per l'esecuzione dell'istruzione $h-$esima, perché non so esattamente quante operazioni macchina servano per eseguire l'istruzione (colonna "costo");
- Inoltre, effettuando una valutazione nel **caso pessimo**, per ogni $h-$esima istruzione si specifica il **massimo numero di volte** che questa viene eseguita (colonna "# Volte").

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

	Costo	# Volte
item $min = A[0]$	c_1	1
for $i = 1$ **to** $n-1$ **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

N.B: per quanto riguarda il ciclo for, è giusto scrivere che viene eseguito n volte, poiché anche la $n-1$esima volta la riga viene eseguita prima di capire che la condizione non è rispettata. Infatti come si può notare, le istruzioni al suo interno vengono eseguite $n-1$ volte.

Dunque, il tempo di calcolo $T(n)$ di `min()` si ottiene sommando il prodotto del costo di ciascuna istruzione per il numero di volte che e stata eseguita:

$T(n) = c_1 + c_2(n) + c_3(n-1) + c_4(n-1) + c_5 = (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) = an + b$
(Posso scrivere $an + b$ perché non si conosce il valore dei costi da $c_1 ... c_5$).

Osservazioni sull'identificazione del caso

Per distinguere caso ottimo, medio e pessimo, occorre capire come varia il numero di operazioni dell'algoritmo al variare dell'input.
Nel caso della funzione `min()`, il numero di iterazioni del ciclo e il numero di confronti eseguiti sono **indipendenti dai valori dell'input**: il `for` viene sempre eseguito n volte e il confronto viene sempre effettuato $n-1$ volte.
Di conseguenza, la funzione presenta un tempo di esecuzione $T(n)$ lineare sia nel caso pessimo sia nel caso medio.

1.3.2 Tempo di calcolo della funzione `binarySearch()` (ricorsiva)

In quest'altro caso si considera la funzione `binarySearch()` che si occupa di ricercare **la posizione** di un elemento all'interno di una sequenza ordinata, memorizzata in un vettore A. La logica che sta dietro la **ricerca binaria**, è la stessa che viene utilizzata per la ricerca di un nodo negli alberi binari: ogni volta che viene richiamata la funzione in modo ricorsivo, si elimina metà del vettore (**ricerca dicotomica** - capitolo **??**).

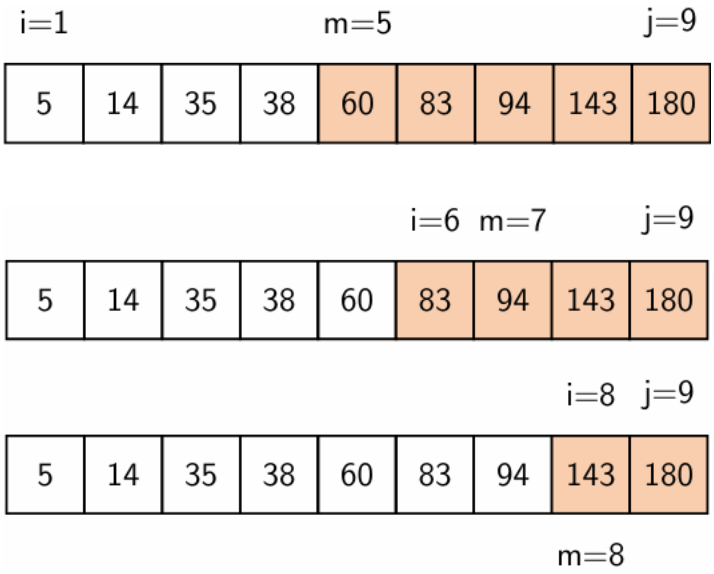
int binarySearch(ITEM[] A, ITEM v, **int** i, **int** j)

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

A differenza del caso precedente (capitolo 1.3.1), l'algoritmo esegue porzioni di codice differenti a seconda dei valori di input (*i* e *j*), che sono rispettivamente, l'**indice iniziale** del vettore e l'**indice finale** del vettore. Per avere una valutazione del tempo di calcolo si andrà a valutare il **caso pessimo** di **entrambe le porzioni** di codice, inserendo due colonne etichettate con "#", che indicano quante volte ciascuna riga viene eseguita :

- $i > j$: questa porzione di codice esegue direttamente la condizione di chiusura poiché se *i* è maggiore di *j* si sta indicando un insieme di elementi nullo in cui la posizione dell'elemento non può esistere.
  In questo caso si può vedere come, nella colonna $\#(i > j)$, vengono eseguite solo le righe con costo c_1 e c_2, poiché subito dopo la ricorsione termina, di conseguenza tutte le altre righe vengono eseguite 0 volte. Dunque il tempo di calcolo sarà dato da: $T(n) = c_1 + c_2 = c$, dove n è zero perché non esiste una grandezza (n) specifica per il vettore.
- $i < j$: in quest'altra porzione di codice il vettore viene suddiviso in due parti **sinistra** e **destra**. La parte sinistra ha grandezza $[(n-1)/2]$, mentre la parte destra $[n/2]$.

Il caso pessimo prevede la ricerca di un elemento **maggiore del massimo contenuto** nel vettore, dunque v sarà sempre maggiore di $A[m]$ e l'algoritmo sceglierà sempre la parte di vettore più grande ($[n/2]$).



La ricerca di un elemento non presente all'interno del vettore, causa l'esecuzione ricorsiva delle istruzioni con costo: $c_3, c_4, c_6, c_7 + T(n/2)$.
Inoltre, ad ogni ricorsione viene esplorata la metà destra, scartatando metà degli elementi rimanenti, aggiornando gli indici di inizio (i), fine (j) e dell'elemento mediano del vettore (m), fino ad esaurire tutti gli elementi. Il costo della funzione è quindi: $T(n) = c_1 + c_2 + c_4 + c_6 + c_7 + T(\frac{n}{2}) = d + T(\frac{n}{2})$ dove d rappresenta la somma da $c_1 ... c_7$.

Questo significa che l'equazione non è definita in modo semplice da un solo valore, ma da una **relazione di ricorrenza** (capitolo 1.5). Una tecnica che viene utilizzata per risolvere le relazioni di ricorrenza consiste nel **produrre una catena di uguaglianze** ottenute per sostituzioni successive.

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

Infatti possiamo scrivere che $T(n) = T(n/2) + d = T(n/4) + 2d = ... = T(n/2^k) + kd$, da cui possiamo ricavare che $\frac{n}{2^k} \Rightarrow k = log\ n$.
Andando avanti con la ricorrenza si arriverà ad un punto in cui $T(n/2^k) + kd = T(1) + kd = [T(0) + d] + kd = T(0) + d(k + 1) = c + kd + d = d\ log\ n + (c + d)$.

1.4 Ordini di complessità

Dopo aver analizzato gli algoritmi al capitolo 1.3 sono state ottenute due **funzioni di complessità**, con una **serie di parametri** che **non si è in grado di determinare**.
Questa difficoltà viene aggirata utilizzando il concetto di **crescita asintotica**.

Concetto di crescita asintotica

Quando analizziamo un algoritmo, non ci interessa tanto il tempo preciso di esecuzione, ma **come cresce questo tempo** quando la dimensione dell'input n diventa molto grande.
- ***Crescita lineare***: `min`: $an + b \Rightarrow O(n)$
- ***Crescita logaritmica***: `BinarySearch()`: $d\ log\ n + (c + d) \Rightarrow O(log\ n)$

Questo concetto è utile perché permette di **confrontare algoritmi diversi** senza **preoccuparsi** delle costanti (che dipendono dall'hardware o dall'implementazione).
Permette di capire **quale algoritmo sarà più efficiente** su input grandi, anche se su input piccoli può sembrare più lento.

Esempio intuitivo: Supponiamo di avere due algoritmi.
- ***Algoritmo A***: richiede $100n$ operazioni $\rightarrow$ complessità $O(n)$;
- ***Algoritmo B***: richiede n^2 operazioni $\rightarrow$ complessità $O(n^2)$

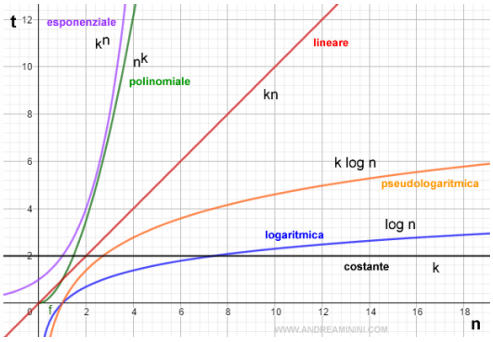
→ Per $n = 10$, A fa 1000 operazioni, mentre B ne fa 100;
→ Per $n = 100$, A fa 100.000 operazioni, mentre B ne fa $1.000.000$;
Quindi asintoticamente A è il migliore, anche se su input piccolo B sembrava il migliore.

1.4.1 Principali classi di efficienza asintotiche

Nella seguente tabella vengono mostrati il numero di passi necessari per compleatre l'algoritmo in base alla complessità e alla dimensione dell'input.

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	logaritmico
n	10	100	1.000	10.000	lineare
$n \log n$	30	664	9.965	132.877	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	1.024	10^{30}	10^{300}	10^{3000}	esponenziale

(a)



(b)

- **Complessità *logaritmica***: Tipicamente, il risultato di ridurre le dimensioni del problema di un fattore costante ad ogni iterazione. **N.B**: un algoritmo in questa classe di effcienza non può tenere conto di tutto il suo input, altrimenti avrebbe effcienza lineare;
- **Complessità *lineare***: Algoritmi che e ettuano un numero costante di iterazioni sull input (ad esempio una ricerca sequenziale);
- **Complessità *loglineare* (o superlineare)**: Algoritmi che necessitano di effettuare almeno una scansione completa dell'input, ma che riducono di un fattore costante le iterazioni intermedie (ad esempio gli algoritmi divide-et-impera);
- **Complessità *quadratica***: Tipica degli algoritmi che sono basati su due iterazioni annidate. Ad esempio, alcuni algoritmi di ordinamento che effettuano operazioni su matrici $n \cdot n$;
- **Complessità *cubica***: Tipica degli algoritmi che sono basati su tre iterazioni annidate, diversi algoritmi di algebra lineare ricadono in questa classe;
- **Complessità *esponenziale***: Algoritmi che e ettuano ricerca sui sottoinsiemi di un insieme di n elementi;
- **Complessità *fattoriale***: Algoritmi che e effettuano ricerca su permutazioni di un insieme di n elementi.

1.4.2 Notazioni asintotiche

Per **descrivere la crescita asintotica** di un algoritmo, quindi come cresce il tempo di esecuzione dell'algoritmo al crescere dell'input n, vengomo utilizzate delle **notazioni**.

Notazione Big-O (O grande)

Descrive il **limite superiore** della crescita di un algoritmo.
- Garantisce che il tempo di calcolo non esploderà oltre una certa funzione;
- Di conseguenza, viene utilizzata per **descrivere il caso peggiore**.

Esempio: se un algoritmo è $O(n)$, significa che al crescere dell'input non farà mai più di un numero di operazioni proporzionale a n.

Notazione Omega (Ω grande)

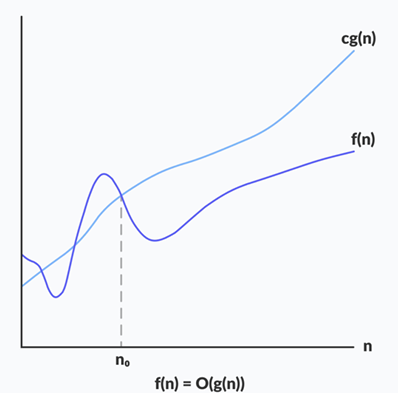
Descrive il **limite inferiore** della crescita di un algoritmo.
- Indica il lavoro minimo che l'algoritmo deve svolgere, anche nel caso migliore;
- Serve a capire che **sotto una certa soglia di complessità non si può scendere**.

Esempio: se un algoritmo è $\Omega(n)$, significa che anche nel caso più favorevole deve comunque guardare almeno n elementi.

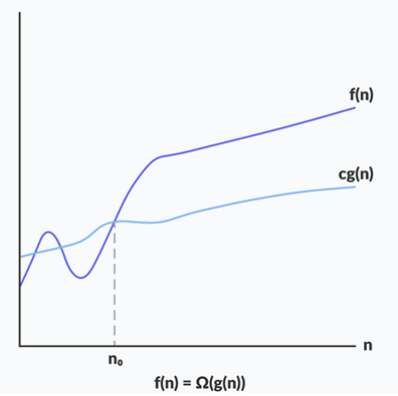
Notazione Theta (Θ grande)

Descrive i due **limiti** (**inferiore e superiore**) della crescita di un algoritmo.
- Indica la **crescita reale** dell'algoritmo;
- Torna utile quando sappiamo che un algoritmo cresce esattamente come una certa funzione, permettendo di **classificarlo con precisione**.

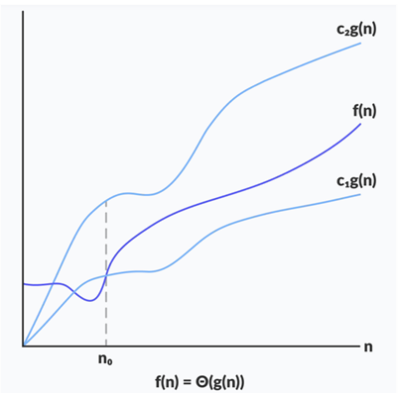
Esempio: se un algoritmo è $\Theta(nlogn)$, significa che cresce proprio in quel modo: non più veloce, non più lento.



(a) Notazione O grande



(b) Notazione Ω grande



(c) Notazione Θ grande

1.5 Le ricorrenze

Quando un algoritmo contiene una **chiamata ricorsiva a se stesso** il suo tempo di esecuzione spesso può essere descritto attraverso **una ricorrenza**.

Cos'è una ricorrenza?

Una **ricorrenza** è un'equazione che descrive una funzione (tipicamente il tempo di esecuzione $T(n)$) in termini del suo stesso valore calcolato su input più piccoli.

Le ricorrenze nascono quando un algoritmo ricorsivo **divide il problema in sottoproblemi** più piccoli e **richiama se stesso**.
Infatti, come è stato visto al capitolo 1.3.2, l'algoritmo `binarySearch()` dimezza il problema ad ogni passo tramite la seguente ricorrenza: $T(n) = T(\frac{n}{2}) + d$, che abbiamo visto avere complessità logaritmica $O(log\ n)$.
Quindi, è possibile esprimere il tempo di esecuzione viene espresso come **la somma di**:
- Il **costo** per **dividere o combinare** il problema (nel caso precedente d);
- Il **costo** per le **chiamate ricorsive** su sottoproblemi più piccoli (nel caso precdente $T(\frac{n}{2})$).

Per risolvere le ricorrenze è possibile **utilizzare tre metodi** differenti: metodo di **sostituzione**, metodo dell'**esperto** o metodo dell'**albero di ricorsione**.

1.5.1 Metodo di sostituzione (o per tentativi)

Idea di base

Il metodo della sostituzione consiste nell'**ipotizzare la forma della soluzione** per poi utilizzare l'**induzione** matematica per **dimostrare che la soluzione ipotizzata** funziona.

Il metodo di sostituzione puo essere usato per determinare il limite inferiore o superiore di una ricorrenza, e **la sua applicazione ha senso** solo nei casi in cui è "**facile**" **immaginare** la **forma** della soluzione.
Ad esempio, una prima ipotesi può essere effettuata nei casi in cui l'algoritmo **rispecchia una delle descrizioni** di complessità (logaritmica, lineare, loglineare, ecc...) - Capitolo 1.4.1.

Una volta fatta l'ipotesi, si procede a produrre una **catena di uglianze** ottenute per **sostituzioni successive** (come già fatto al capitolo 1.3.2) cercando di arrivare all'ipotesi effettuata in precedenza.

Osservazioni

È importante notare che non esiste un metodo generale per formulare l'ipotesi della soluzione corretta di una ricorrenza, infatti se quest'ultima è simile ad una già vista, **ha senso provare una soluzione analoga**.

1.5.2 Metodo dell'esperto

Idea di base

Il metodo dell'esperto (Master Theorem) è una "***formula pronta***" per un'**intera famiglia di ricorrenze** tipiche, che dividono un problema di dimensione n in a sottoproblemi di dimensione n/b.

La seguente formula $T(n) = aT(n/b) + f(n)$ dove $a \geqslant 1$ e $b > 1$ fornisce subito il risultato:
- Ogni $a-$esimo sottoproblema viene risolto nel tempo $T(n/b)$;
- Il costo per dividere il problema e combinare i risultati dei sottoproblemi è descritto da $f(n)$

Dunque, questa metodologia di soluzione per le ricorrenze **risulta perfetta solo per algoritmi classici** come mergesort, binary search, quicksort, ecc...

Proprio perché viene utilizzata per una famiglia di algoritmi tipici di cui si conosce l'andamento reale del caso pessimo $\Theta(...)$, è possibile **individuare tre casi** in cui l'andamento dell'algoritmo può ricadere:
Date le costanti intere $a \geqslant 1$ e $b \geqslant 2$ e le costanti reali $c > 0$ e $\beta \geqslant 0$. Sia $T(n)$ data dalla relazione di ricorrenza:

$$T(n) = \begin{cases} c & se\ n \leqslant 1 \\ aT(n/b) + cn^{\beta} & se\ n > 1 \end{cases} \quad con\ \alpha = \frac{log\ a}{log\ b} = log_b a : \quad T(n) = \begin{cases} \Theta(n^{\alpha}) & \alpha > \beta \\ \Theta(n^{\alpha} logn) & \alpha = \beta \\ \Theta(n^{\beta}) & \alpha < \beta \end{cases}$$

Esempio del *Caso 1*
Immaginiamo di avere la ricorrenza $T(n) = 9T(n/3) + n$, si ha dunque:
- $a = 9, b = 3$
- $\alpha = log_3 9 = 2$
- $\beta = 1$

Quindi $\alpha > \beta \Rightarrow T(n) = \Theta(n^\alpha) = \Theta(n^2)$

Esempio del *Caso 2*
Immaginiamo di avere la ricorrenza $T(n) = T(n/3) + 1$, si ha dunque:
- $a = 1, b = 3$
- $\alpha = \frac{log1}{log3} = 0$
- $\beta = 0$

Quindi $\alpha = \beta \Rightarrow T(n) = \Theta(n^\alpha log\ n) = \Theta(log\ n)$

Esempio del *Caso 3*
Immaginiamo di avere la ricorrenza $T(n) = 3T(n/4) + n$, si ha dunque:
- $a = 3, b = 4$
- $\alpha = \frac{log4}{log3} = 0.793$
- $\beta = 1$

Quindi $\alpha < \beta \Rightarrow T(n) = \Theta(n^\beta) = \Theta(n)$

1.5.3 Metodo dell'albero di ricorsione (analisi per livelli)

Idea di base

La **ricorsione** viene immaginata **come un albero**: ogni **nodo è un sottoproblema** con un determinato costo, e **ogni livello** dell'albero rappresenta **un passo** della ricorsione.

Quindi, si calcola il **costo di ogni livello** (somma dei costi dei singoli nodi per quel livello), e poi **si sommano tutti i livelli**. Possiamo immaginarlo come un processo di questo tipo:
- ***Livello 0***: abbiamo l'espressione originale;
- ***Livello 1***: espressione a cui è stata applicata un'espansione;
- ***Livello 2***: vengono applicate due espansioni;
- ... si continua fino al caso base.

Livello	Espressione	Costo
0	n^2	$4^0 n^2$
1	$\frac{n^2}{2^2}\ \frac{n^2}{2^2}\ \frac{n^2}{2^2}\ \frac{n^2}{2^2}$	$4^1 \frac{n^2}{2^2}$
2	$\frac{n^2}{2^4}\ \frac{n^2}{2^4}\ \dots\ \frac{n^2}{2^4}\ \frac{n^2}{2^4}$	$4^2 \frac{n^2}{2^4}$
	...	
i	$\frac{n^2}{2^{2i}}\ \frac{n^2}{2^{2i}}\ \dots\ \frac{n^2}{2^{2i}}\ \frac{n^2}{2^{2i}}$	$4^i \frac{n^2}{2^{2i}}$
	...	
h	$T(1)\,T(1)\ \dots\ T(1)\,T(1)$	4^h

Per comprendere meglio il concetto si consideri la ricorrenza:

$$T(n) = \begin{cases} 1 & se\ n = 1 \\ 4T(n/2) + n^2 & n = 2^h,\ h > 0 \end{cases}$$

Dunque, il **costo complessivo** di ciascun livello dell'albero (escluso l'ultimo che è il caso base) è sempre n^2: i sottoproblemi sono più piccoli ma più numerosi. Invece, $4T(n/2)$ è il costo per la chiamata ricorsiva che genera 4 sottoproblemi di dimensione $n/2$.

- ***Livello 0* (radice)**
  - Si ha un solo problema di dimensione n.
- ***Livello 1***
  - Il problema si divide in 4 sottoproblemi (coefficiente 4);
  - Il costo di ciascuno è $n^2/4$;
  - Ognuno ha dimensione $n/2$;
  - Costo totale del livello $4 \cdot (n^2/4) = n^2$
- ***Livello 2***
  - Ogni sottoproblema del livello 1 si divide di nuovo in $4 \rightarrow$ in totale $4^2 = 16$ sottoproblemi;
  - Il costo di ciascuno è $n^2/16$;
  - Ognuno ha dimensione $n/4$;
  - Costo totale del livello $16 \cdot (n^2/16) = n^2$

Quindi, seguendo il pattern, il numero di sottoproblemi cresce ad ogni livello di 4^i e la dimensione di ognuno diminuisce di $n/2^i$.

In questo caso specifico è facile capire la complessità della ricorrenza, poiché **il numero di sottoproblemi cresce esponenzialmente** (4^i), ma allo stesso momento la loro dimensione cala esponenzialmente ($n/2^i$) e questo permette al costo totale di ogni livello di rimanere costante (n^2), ma ripetuto per tutti i livelli. Dunque, la ricorrenza ha complessità $O(n^2\ log\ n)$.

Osservazioni

Grazie al fatto che la ricorsione viene suddivisa il livelli, questo approccio risulta molto utile per capire intuitivamente dove si concentra il costo (in alto, in basso o se è distribuito su tutti i livelli)

1.6 Ordinamento

Dato un generico problema, è possibile progettare un gran numero di **algoritmi differenti** per la sua risoluzione, ognuno caratterizzato dal proprio tempo di calcolo: alcuni molto lenti, altri molto veloci.

- Gli algoritmi con **complessità esponenziale** (tipo 2^n) **non sono accettabili** come soluzione nel momento in cui si lavora con **input grandi**, a meno che non sia possibile dimostrare che il problema posto sia inerentemente difficile;
- Invece, realizzando un'algoritmo di **complessità polinomiale** (tipo n^2 o $n \cdot logn$), si ha già fatto un buon lavoro, ma si cerca comunque di "*abbassarne la complessità*".

Obbiettivo

In questi casi l'**obbiettivo** principale è quello di valutare gli algoritmi in base alla **tipologia dell'input**. Ovviamente gli algoritmi "*migliori*" sono quelli che garantiscono tempi di esecuzione accettabili anche su input molto grandi.

Infatti, in alcuni casi, gli **algoritmi** si **comportano diversamente** in base alle **caratteristiche dell'input** e conoscere in anticipo tali caratteristiche permette di scegliere il miglior algoritmo per quella determinata situazione.

Il **problema dell'ordinamento** è un buon esempio per mostrare questi concetti, proprio perché **comparendo in tanti contesti** e avendo **soluzioni diverse** è perfetto per scegliere l'implementazione migliore in base all'input.

Problema dell'ordinamento (sorting)

Dato un vettore di n elementi, il problema dell'ordinamento prevede la **permutazione del vettore** per fare in modo che i suoi elementi compaiano in **ordine non decresente**.

Se si volesse utilizzare un **approccio** "***demente***", si potrebbero generare tutte le possibili permutazioni fino a che non se ne trova una già ordinata, in questo caso:

- Per verificare che un vettore A sia ordinato, basta un ciclo `for`, quindi richiede $O(n)$ tempo;
- Il numero di permutazioni possibili è $n!$.

Dunque, procedendo il questo modo si avrebbe una complessità $O(n \cdot n!)$, ovvero una **complessità superpolinomiale**. Per evitare ciò, l'approccio migliore è quello di utilizzare degli **algoritmi di ordinamento**, decisamente più adatti a questa tipologia di problema.

1.6.1 Selection Sort

Il selection sort è un semplice algoritmo polinomiale che è basato sulla proprietà che in una sequenza ordinata, il primo elemento ha valore minimo.

Algoritmo selection sort

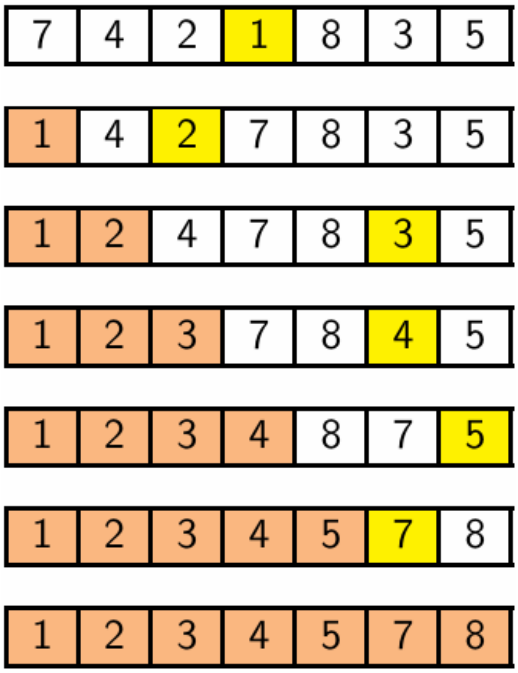
In un **algoritmo selection sort** si cerca il minimo e si scambia tale elemento con quello nella prima posizione della **parte non ordinata del vettore**, riducendo il problema agli $n - 1$ restanti valori.

Dato un input del tipo $A = \{7, 4, 2, 1, 8, 3, 5\}$ con $n = 7$, l'algoritmo `selectionSort()` si comporta nel seguente modo:

```
Algorithm selectionSort(Item[] A, int n)
1: for i = 1 to n − 1 do
2:     int j = min(A, i, n)
3:     A[i] ↔ A[j]
4: end for
```

```
Algorithm int min(Item[] A, int k, int n)
1: int min = k
2: for h = k + 1 to n do
3:     if A[h] < A[min] then
4:         min = h
5:     end if
6: end for
7: return min
```

(a)



(b)

Complessità del `selectionSort()`

In questo particolare algoritmo **non importa quale sia l'ordine iniziale dell'input** dato. Questo perché lo **spostamento di un valore** ha sempre **costo costante**.

L'algoritmo `selectionSort()` esegue un ciclo esterno che va da $i = 1$ a $n - 1$, e ad ogni iterazione, chiama la funzione `min()` su un sottoarray da i a n.
La funzione `min()` esegue un ciclo interno che fa $n - i$ confronti, quindi sempre meno confronti man mano che il vettore si riordina.
Sapendo che la somma dei primi k numeri naturali è $1 + 2 + ... + k = \frac{k(k+1)}{2}$ possiamo dire che:

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

Questo cresce come n^2, quindi la complessità è quadratica $O(n^2)$, anche se l'array è già ordinato, perché deve **comunque cercare il minimo** nel sottoarray, anche se è già al posto giusto.

1.6.2 Insertion sort

L'algoritmo `insertionSort()` è un'algoritmo efficiente per ordinare piccoli insiemi di elementi. Si basa sul principio di ordinamento di una mano di carte da gioco.

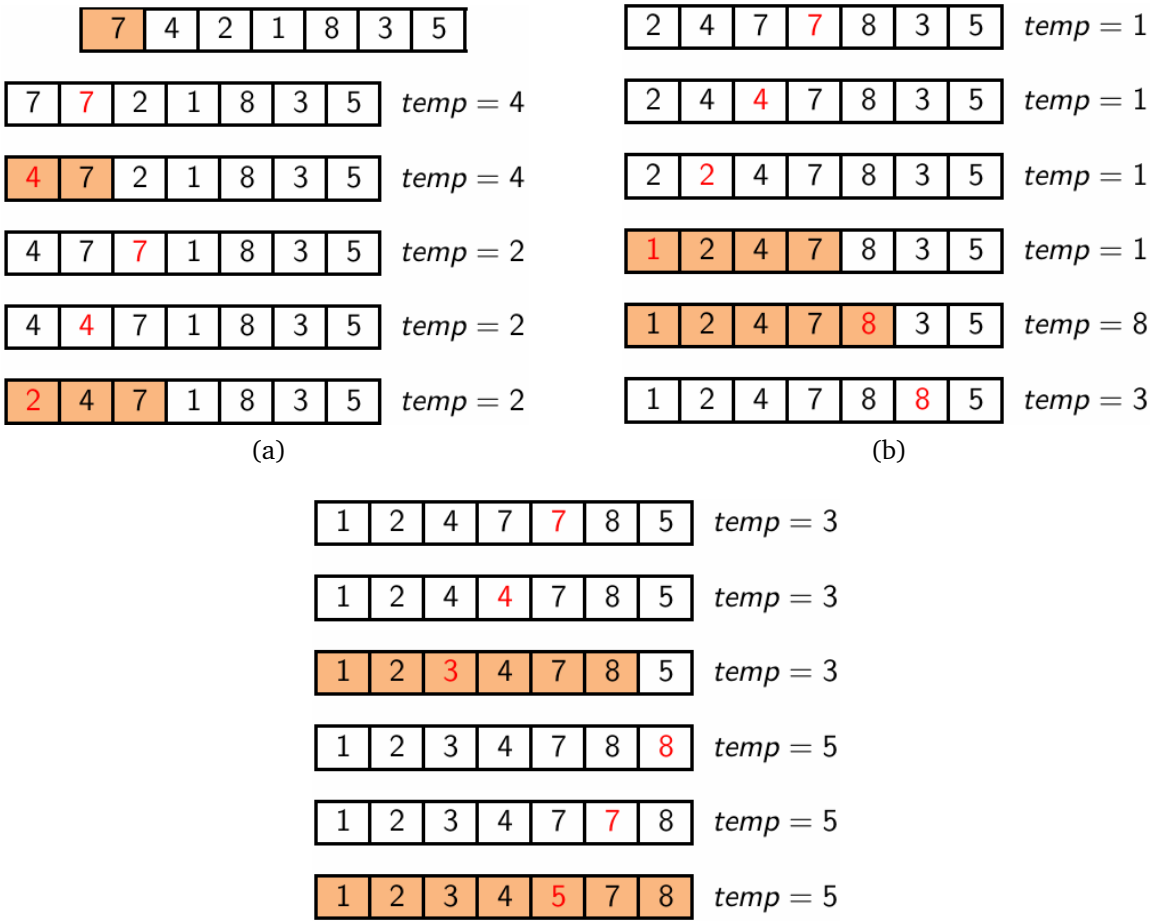
Algoritmo insertion sort

Per seguire l'analogia, si considerino le carte una per volta, ad esempio, da sinistra verso destra: ogni volta che viene considerata una nuova carta, si inserisce nella posizione giusta rispetto alle altre carte gia considerate e ordinate, traslando di una posizione verso destra tutte le carte maggiori.

```
Algorithm insertionSort(Item[] A, int n)
1: for i = 2 to n do
2:     Item temp = A[i]
3:     int j = i
4:     while j > 1 and A[j − 1] > temp do
5:         A[j] = A[j − 1]
6:         j = j − 1
7:     end while
8:     A[j] = temp
9: end for
```

Immaginiamo di avere un input del tipo $A = \{2, 3, 1, 4, 7, 3\}$.
L'**idea di base** è la seguente: se si prende solo il primo elemento $[2]$, in se per se è già ordinato, potrebbe anche non essere la sua posizione giusta, ma preso come sottovettore è ordinato.
Vado poi a vedere il valore successivo $[3]$ e lo confronto con il valore $[2]$ (che è l'ultimo del vettore ordinato): $[3] > [2]$ quindi la parte iniziale del vettore è ordinata {2, 3, ...}.
Andando avanti si troverà $[1]$, più piccolo sia di $[2]$ che di $[3]$. L'elemento minore viene messo in una variabile temporantea (`temp`), mentre gli elementi della parte già ordinata del vettore vengono traslati (partendo dal più grande) uno ad uno verso destra di una posizione, sovrascrivendo l'elemento alla propria destra. A questo punto all'inizio del vettore si è creato uno spazio vuoto $[...]$ (dovuto alla traslazione della parte ordinata del vettore) su cui andrà inserita la variabile `temp` contenente l'elemento precedentemente sovrascritto, nonché il più piccolo elemento del vettore esplorato fino a quel momento.

Ovviamente, nel caso in cui l'elemento da inserire vada posizionato nel mezzo del vettore ordinato, durante la traslazione dei singoli elementi ci si fermerebbe nel punto in cui `temp` sia maggiore dell'elemento `A[i]`. Ad esempio, si consideri un input del tipo $A = \{7, 4, 2, 1, 8, 3, 5\}$.



(a) (b)

(c)

Per la logica che sta dietro al `selection sort()`, anche se l'input iniziale è già ordinato, l'algoritmo andrà a cercare comunque il valore minimo e lo sovrascriverà nella prima posizione della parte non ordinata dell'array, quindi, come detto al capitolo (capitolo 1.6.1), la complessità dell'algoritmo non dipende dall'input di ingresso.

Invece, a differenza del `selection sort()`, la **complessità** dell'`insertion sort()` dipende dalla **disposizione iniziale dei dati in ingresso** poiché, per come è strutturato l'algoritmo, man mano che si scorre l'array si andranno a posizionare i nuovi elementi nella posizione corretta rispetto a tutti gli altri.
Proprio per questa caratteristica, la complessità dell'`insertion sort()` cambia a seconda del caso di studio (capitolo 1.3):

- *Complessità nel caso **pessimo***: il caso peggiore si ha quando l'input A è **ordinato alla rovescia**, ad esempio: $A = \{8, 7, 6, 5, 4, 3, 2, 1\}$.
  In questo caso ad ogni iterazione del ciclo `for`, si entrerebbe anche nel ciclo `while` più interno poiché la variabile `temp` andrebbe spostata in prima posizione, e ciò richiederebbe un numero di spostamenti pari a
  $$\sum_{i=1}^{n-1}(n-i)$$
  Complessità? Dunque la complessità sarebbe uguale a quella del `selection sort()` per la presenza dell'annidamento dei due cicli $\rightarrow O(n^2)$.

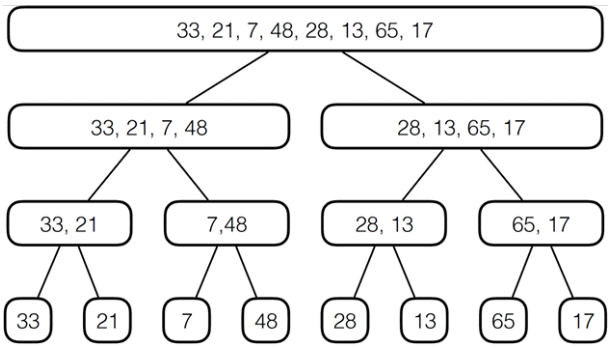
- *Complessità nel caso* ***ottimo***: il caso migliore si ha quando l'input A è una sequenza già ordinata, quindi $A = 1, 2, 3, 4, 5, 6, 7, 8$
  In questo caso, non avverrà mai che `A[j-1]` $[1]$ sia maggiore di `temp` $[2]$ e cosi via..., non permettendo al ciclo `while` di verificarsi. Rimane solo il ciclo esterno, che esegue operazioni di costo **costante**.
  Complessità? In questo caso, la complessità per delle istruzioni costanti vale $O(n)$.

1.6.3 Merge sort

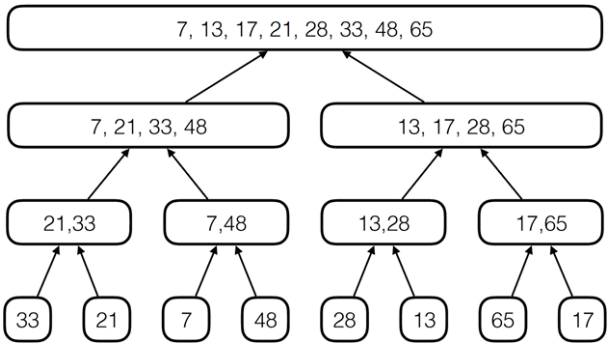
L'algoritmo `MergeSort()` è basato sulla tecnica **divide et impera**.

Algoritmo merge sort

- **Divide**: spezza il vettore di n elementi in due sottovettori di $n/2$ elementi;
- **Impera**: chiama `MergeSort()` ricorsivamente sui due sottovettori;
- **Combina**: una volta ottenuti singoli valori, questi vengono riuniti (merge) in modo ordinato, fino a riottenere i due sottovettori iniziali ordinati.



(d) Partizionamento dei dati



(e) Fusione dei dati

```
Algorithm mergeSort(Item[]A, int first, int last)
1: if first < last then
2:     int mid = ⌊(first + last)/2⌋
3:     mergeSort(A, first, mid)
4:     mergeSort(A, mid+1, last)
5:     merge(A, first, last, mid)
6: end if
```

La prima parte dell'algoritmo `mergeSort()` si occupa di effettuare le chiamate ricorsive per la suddivisione del vettore non ordinato fino ai singoli valori (come in figura *d*). Prendiamo in considerazione il vettore: $A = [4, 3, 2, 1]$.
Nella *riga 2* viene calcolata la metà del vettore (se il vettore è dispari si andrà per eccesso). In questo caso, il vettore viene diviso in parte sinistra $[4, 3]$ e parte destra $[2, 1]$.

Prima ricorsione (riga 2)
Una volta che il vettore è stato diviso in due parti, la prima istruzione ricorsiva (*riga 3*) richiama la funzione `mergeSort(A, first, mid)` per poter lavorare sulla parte sinistra, ossia $[4, 3]$. Dunque, la *riga 2* viene rieseguita e il sottovettore viene ulteriormente suddiviso in:
- parte sinistra $[4]$;
- parte destra $[3]$.

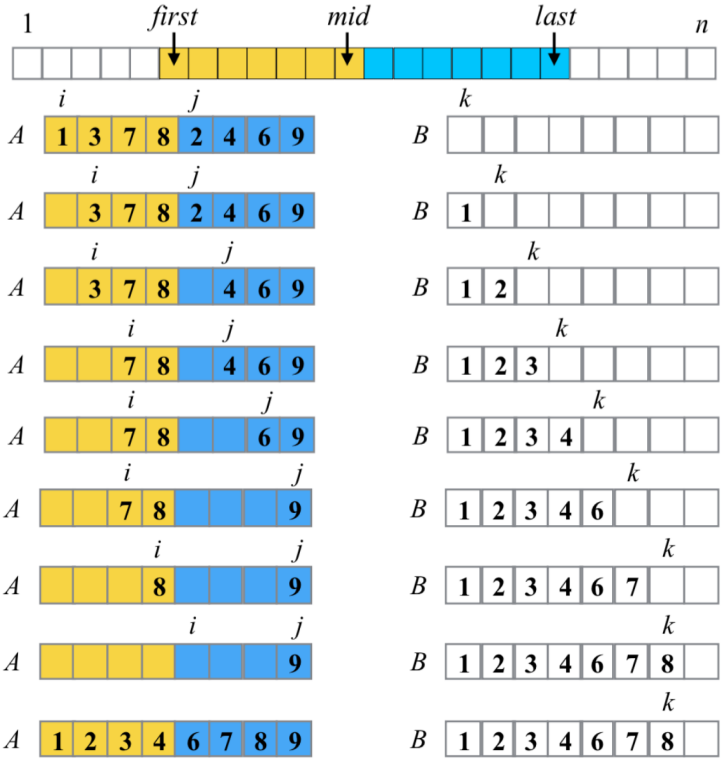
Subito dopo, le istruzioni ricorsive vengono tentate nuovamente, ma:
- `mergeSort(A, first, mid)` non prosegue poiché il sottovettore ha un solo elemento $[4]$;
- anche, `mergeSort(A, mid+1, last)` non prosegue perché anche $[3]$ è un singolo elemento.

Raggiunto il caso base per entrambi i sottovettori, è possibile richiamare `merge()` per combinarli e ottenere il vettore ordinato $[3, 4]$.

Seconda ricorsione (riga 3)
Terminata la ricorsione sulla parte sinistra del vettore iniziale, il controllo ritorna alla chiamata precedente, che ora può eseguire l'istruzione ricorsiva successiva (*riga 4*) sulla parte destra, utilizzando `mergeSort(A, mid+1, last)`.
Il procedimento è analogo al precedente: la *riga 2* suddivide il sottovettore destro in $[2]$ e $[1]$, mentre le istruzioni ricorsive (*righe 3* e *4*) non proseguono perché entrambi i vettori hanno un solo elemento. Infine viene richiamata `merge()` per ordinare i due elementi e ottenere $[1, 2]$.

Merge finale (riga 4)
Terminate entrambe le ricorsioni, significa che i due sottovettori originari sono ora ordinati. L'ultima istruzione rimasta della chiamata principale è dunque la `merge()` finale, che combina i due sottovettori ordinati per ottenere un unico vettore ordinato (come illustrato in figura *e*).

Nello specifico, una generica operazione di `merge()` funziona in questo modo:



N.B: *first, last e mid* sono tali che $1 \leqslant first \leqslant mid \leqslant last \leqslant n$.

Bisogna specificare che il merge agisce su dei sottovettori già ordinati: infatti il primo `merge()` viene effettuato sui singoli elementi, e da quì, ogni volta che si effettua un `merge()` si avranno sempre dei sottovettori ordinati, $A[first...mid]$ e $A[id + 1...last]$.

Per fondere le due metà ordinate, la procedura `merge()` si avvale di un vettore di appoggio B, utilizzato come parametro globale.
Vengono qundi utilizzati tre indici i, j e k per scandire, rispettivamente, $A[first..mid]$, $A[mid + 1...last]$ e $B[start...end]$. Ad ogni passo, sono confrontati gli elementi $A[i]$ e $A[j]$, il minore viene copiato in $B[k]$, e vengono incrementati di una posizione k e l'indice dell'elemento che risulta minore.

Il procedimento viene iterato fino a che una delle due metà è esaurita ($A[first..mid]$ oppure $A[mid + 1...last]$). Non appena una delle due metà si svuota per prima, è possibile proseguire l'ordinamento in modi differenti:

- Se la **prima metà è stata esaurita per prima** ($i > mid$) gli eventuali elementi $A[j..end]$ della metà non scandita si trovano già nella posizione corretta per l'ordinamento.
  Dunque, non è necessario spostare gli elementi non scanditi in A nel vettore di appoggio B, ma piuttosto, spostare tutti gli elementi già ordinati da B ad A;
- Se invece **si svuota prima la seconda parte**, gli elementi non scanditi $A[i...mid]$ della prima metà vengono subito spostati nelle ultime posizioni $A[k...last]$ che competono loro nell'ordinamento finale. Infine, la posizione $B[first...k - 1]$ è ricopiata in $A[first...k - 1]$,

ottenendo così $A[first...last]$.

Il codice per il funzionamento del `merge()` è il seguente, illustrato in figura 34.

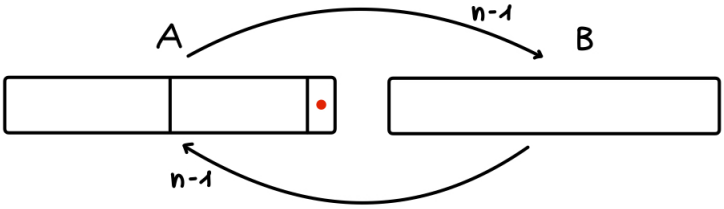
Figure 34: Pseudocodice della funzione `merge()`

Algorithm **merge**(Item[]A, **int** first, **int** last, **int** mid)

```
1: int i, j, k, h
2: i = first; j = mid + 1; k = first
3: while i ≤ mid and j ≤ last do
4:     if A[i] ≤ A[j] then
5:         B[k] = A[i]; {N.B.: B è un vettore di appoggio usato come parametro globale}
6:         i = i + 1
7:     else
8:         B[k] = A[j];
9:         j = j + 1
10:    end if
11:    k = k + 1
12: end while
13: j = last
14: for h = mid downto i do
15:    A[j] = A[h]
16:    j = j − 1
17: end for
18: for j = first to k − 1 do
19:    A[j] = B[j]
20: end for
```

Per capire quale sia la complessità di `mergeSort()` dobbiamo prima trovare separatamente la complessità di `merge()`.

Nel **caso pessimo** ogni valore in A deve essere trasferito in B, e questo accade quando rimangono tutti tranne un solo elemento della sottoparte destra di A. Quindi come illustrato nell'immagine, in A è rimasto un singolo valore che è già nella sua posizione corretta, gli altri $n-1$ valori sono stati trasferiti in B. A questo punto andranno tutti riportati in A, e anche in questo caso l'operazione ha costo $n-1$. Possiamo quindi dire che la complessità di `merge()` è $O(n)$ perché il **numero di operazioni cresce in modo lineare** rispetto al **numero totale di elementi** nei due sottovettori.



Osservazione sulla complessità di `merge()`

Il fatto che la complessità di `merge()` sia $O(n)$ è anche intuibile dal momento che la funzione non contiene cicli annidati: il ciclo principale viene eseguito al massimo $n-1$ volte e ogni iterazione ha costo costante. Ne deriva quindi un costo complessivo lineare.

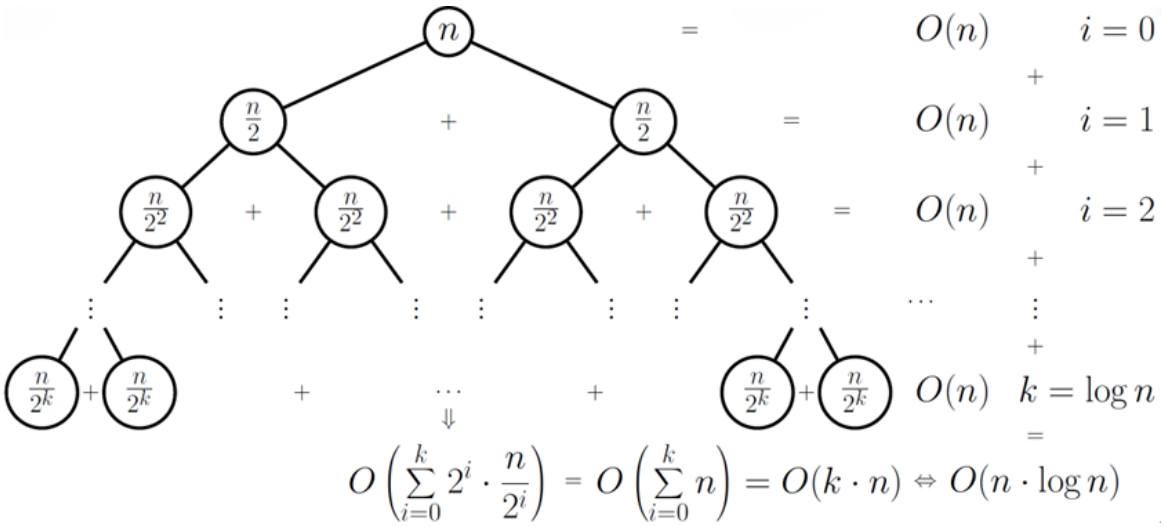
A questo punto l'analisi della complessità si sposta sul capire quale sia la complessità dell'**intero algoritmo**. Quando analizziamo il `mergeSort()` vengono effettuate le seguenti operazioni:

1. Viene preso un array di dimensione n;
2. Viene diviso in **due metà**: una di dimensione $n/2$ e l'altra di dimensione $n/2$;
3. Si applica ricorsivamente il `mergeSort()` su entrambe le metà.

Quindi per ordinare l'array completo:
- Per ordinare la prima metà $\rightarrow$ costo $T(n/2)$;
- Per ordinare la seconda metà $\rightarrow$ costo $T(n/2)$.

Sommando i termini si ottiene: $T(n) = T(n/2) + T(n/2) +$ *costo del merge* $= T(n) = 2T(n/2) + O(n)$, dove:
- $2T(n/2)$ è il tempo per ordinare le due metà;
- $O(n)$ è il tempo per fonderle con `merge()`.

Dopo aver ottenuto l'equazione del costo dell'algoritmo, per **trovare la complessità**, possiamo approfittare del fatto che `mergeSort()` sia un algoritmo ricorsivo e utilizzare il **metodo dell'albero di ricorsione** visto al capitolo 1.5.3.



Dunque, il **costo complessivo** di ciascun livello dell'albero è sempre n, perché il costo del `merge()` rimane costante per qualsiasi coppia di sottovettori.
Invece, $2T(n/2)$ è il costo per la chiamata ricorsiva che genera 2 sottovettori di dimensione $n/2$ ogni volta che viene richiamata fino ad ottenere i singoli valori.
Il numero di sottovettori cresce esponenzialmente (2^i), ma allo stesso momento la loro dimensione cala esponenzialmente ($n/2^i$) e il costo totale di ogni `merge()` è costante $O(n)$.
Dunque, la ricorrenza ha complessità $O(n\ log\ n)$.

Se invece si volesse utilizzare il **metodo dell'esperto** visto al capitolo 1.5.2, ricordando la formula generale $T(n) = aT(n/b) + cn^{\beta}$, applicata a $T(n) = 2T(n/2) + O(n)$, si avrà:
- $a = 2$, $b = 2$
- $\alpha = log\ 2/log\ 2 = 1$
- $\beta = 1$

Quindi $\alpha = \beta$ e si ricade nel caso 2: $T(n) = \Theta(n^{\alpha} \cdot log\ n) = \Theta(n \cdot log\ n)$