



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

# Appunti di Algoritmi e Strutture Dati

a.a. 2017/2018

Autore:  
Timoty Granziero

Repository:  
<https://github.com/Vashy/ASD-Notes>

22 marzo 2018

## Indice

<b>1</b>	<b>Lezione del 28/02/2018</b>	<b>3</b>
1.1	Problem Solving . . . . .	3
1.2	Cosa analizzeremo nel corso . . . . .	4
1.2.1	Approfondimento sul tempo di esecuzione $T(n)$ . . . .	4
1.3	Problema dell'ordinamento (sorting) . . . . .	4
1.4	Insertion Sort . . . . .	5
1.4.1	Invarianti e correttezza . . . . .	6
<b>2</b>	<b>Lezione del 02/03/2018</b>	<b>7</b>
2.1	Modello dei costi . . . . .	7
2.2	Complessità di IS . . . . .	8
2.2.1	Caso migliore . . . . .	8
2.2.2	Caso peggiore . . . . .	8
2.2.3	Caso medio . . . . .	9
2.3	Divide et Impera . . . . .	9
2.4	Merge Sort . . . . .	9
2.4.1	Invarianti e correttezza . . . . .	11
<b>3</b>	<b>Lezione del 07/03/2018</b>	<b>13</b>
3.1	Approfondimento sull'induzione . . . . .	13
3.1.1	Induzione ordinaria . . . . .	13
3.1.2	Induzione completa . . . . .	13
3.2	Complessità di Merge Sort . . . . .	13
3.3	Confronto tra IS e MS . . . . .	15
<b>4</b>	<b>Lezione dell'08/03/2018</b>	<b>16</b>
4.1	Notazione asintotica . . . . .	16
4.1.1	Limite asintotico superiore . . . . .	17
4.1.2	Limite asintotico inferiore . . . . .	19
4.1.3	Limite asintotico stretto . . . . .	20
4.2	Metodo del limite . . . . .	21
4.3	Alcune proprietà generali . . . . .	21
<b>5</b>	<b>Lezione del 09/03/2018</b>	<b>22</b>
5.1	Complessità di un problema . . . . .	22
5.2	Esempio: limite inferiore per l'ordinamento basato su scambi .	22
5.3	Soluzione di ricorrenze . . . . .	23
5.3.1	Metodo di sostituzione . . . . .	24
<b>6</b>	<b>Lezione del 14/03/2018</b>	<b>26</b>

<b>7</b>	<b>Lezione del 15/03/2018</b>	<b>29</b>
7.1	Master Theorem . . . . .	29
7.1.1	Esercizi (Master Theorem) . . . . .	30
<b>8</b>	<b>Lezione del 21/03/2018</b>	<b>34</b>
8.1	Heapsort . . . . .	34
8.1.1	Max Heap . . . . .	35
	<b>Appendices</b>	<b>37</b>
<b>A</b>	<b>Raccolta algoritmi</b>	<b>37</b>
A.1	Insertion Sort . . . . .	37
A.2	Merge Sort . . . . .	37
A.3	Insertion Sort ricorsivo . . . . .	38
A.3.1	Correttezza di Insertion-Sort( $A, j$ ) . . . . .	38
A.3.2	Correttezza di Insert( $A, j$ ) . . . . .	39
A.4	CheckDup . . . . .	39
A.4.1	Correttezza di DMerge( $A, p, q, r$ ) . . . . .	40
A.5	SumKey . . . . .	40
A.5.1	Correttezza di Sum( $A, key$ ) . . . . .	41
<b>B</b>	<b>Esercizi</b>	<b>43</b>
B.1	Ricorrenze . . . . .	43

# 1 Lezione del 28/02/2018

## 1.1 Problem Solving

1. Formalizzazione del problema;
2. Sviluppo dell'**algoritmo** (focus del corso);
3. Implementazione in un programma (codice).

**Algoritmo** Sequenza di passi elementari che risolve il problema.

Input  $\rightarrow$  **Algoritmo**  $\rightarrow$  Output

*Dato un problema, ci sono tanti algoritmi per risolverlo.*

**e.g.**<sup>1</sup> Ordinamento dei numeri di una Rubrica. L'idea è quella di trovare tutte le permutazioni di ogni numero.

30 numeri: *complessità*  $30! \cong 2 \times 10^{32} ns \Rightarrow$   
 $3^{19}$ anni (con  $ns$  = nanosecondi)

**std::vector** È un esempio nel C++ delle ragioni per cui si studia questa materia. Nella documentazione della STL, sono riportati i seguenti:

- **Random access**: complessità  $O(1)$ ;
- **Insert**: complessità  $O(1)$  ammortizzato.

Il **random access** è l'accesso a un elemento casuale del **vector**.  $O(1)$  implica che l'accesso avviene in tempo costante (pari a 1).

Per **insert** si intende l'inserimento di un nuovo elemento in coda. Avviene in tempo  $O(1)$  ammortizzato: questo perchè ogni N inserimenti, è necessario un resize del vector e una copia di tutti gli elementi nel nuovo vettore (questa procedura è nascosta al programmatore).

---

<sup>1</sup>For the sake of example.

## 1.2 Cosa analizzeremo nel corso

- Tempo di esecuzione;
- Spazio (memoria);
- Correttezza;
- Manutenibilità.

### 1.2.1 Approfondimento sul tempo di esecuzione $T(n)$

- *P Problems*: complessità polinomiale. L'algoritmo è trattabile
- *NP Complete*: problemi NP completi. **e.g**: Applicazione sugli algoritmi di sicurezza. Si basano sull'assunzione che per essere risolti debbano essere considerate tutte le soluzioni possibili.
- *NP Problems*: problemi con complessità (ad esempio) esponenziale/fattoriale. Assolutamente non trattabili.

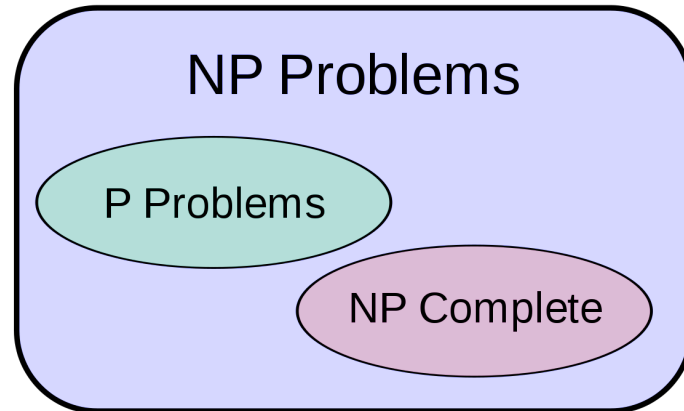


Figura 1: Complessità  $T(n)$ .

## 1.3 Problema dell'ordinamento (sorting)

Input: sequenza di numeri

$$a_0 a_1 \dots a_n;$$

Output: permutazione

$$a'_0 a'_1 \dots a'_n$$

tale che

$$a'_0 \leq a'_1 \leq \dots \leq a'_n$$

Vedremo due algoritmi:

- Insertion Sort;
- Merge Sort.

## 1.4 Insertion Sort

**Insertion Sort** un algoritmo di *sorting incrementale*. Viene applicato naturalmente ad esempio quando si vogliono ordinare le carte nella propria mano in una partita a scala 40: si prende ogni carta a partire da sinistra, e la si posiziona in ordine crescente.

**Astrazione** Prendiamo ad esempio il seguente array:

5	2	8	4	7
---	---	---	---	---

Partiamo dal primo elemento: 5. È già ordinato con se stesso, quindi procediamo con il secondo elemento.

Confronto il numero 2 con l'elemento alla sua sinistra:

$2 \geq 5$ ? No, quindi lo inverte con l'elemento alla sua sinistra, come segue

2	5	8	4	7
---	---	---	---	---

 Key: 

8
---

La key analizzata è 8.

$8 \geq 5$ ? Sì, quindi è ordinato in modo corretto.

2	5	8	4	7
---	---	---	---	---

 Key: 

4
---

La key analizzata è 4.

$4 \geq 8$ ? No, quindi lo sposto a sinistra invertendolo con 8.

$4 \geq 5$ ? No, lo sposto a sinistra invertendolo con 5.

$4 \geq 2$ ? Sì, quindi è nella posizione corretta.

2	4	5	8	7
---	---	---	---	---

 Key: 

7
---

Key analizzata 7.

$7 \geq 8$ ? No, lo sposto a sinistra invertendolo con 8.

$7 \geq 5$ ? Sì, è nella posizione corretta.

Ottengo l'array ordinato:

2	4	5	7	8
---	---	---	---	---

**Algoritmo** Passiamo ora all'implementazione dell'algoritmo, con uno pseudocodice simile a Python<sup>1</sup>

**Input:**  $A[1, \dots, n]$ ,  $A.length$ .

È noto che:  $A[i] \leq key < A[i + 1]$

**Pseudocodice** Segue lo pseudocodice dell'Insertion Sort.

INSERTION-SORT( $A$ )

```
1   $n = A.length$ 
2  for  $j = 2$  to  $n$  // il primo elemento è già ordinato
3       $key = A[j]$  //  $A[1..j - 1]$  ordinato
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

Quando il **while** termina, ci sono due casi:

- $i = 0$ : tutti gli elementi prima di  $j$  sono maggiori di  $key$ ;  $key$  va al primo posto (1);
- $(i > 0)$  and  $(A[i] \leq key)$ :  $A[i+1] = key$ .

#### 1.4.1 Invarianti e correttezza

**for**  $A[1..j-1]$  è ordinato e contiene gli elementi in  $(1, j-1)$  iniziali.

**while**  $A[1..i]A[i+2..j]$  ordinato e  $A[i+2..j] > key$ .

In uscita abbiamo:

- $j = n+1$ ;
- $A[1..n]$  ordinato, come da invariante: vale  $A[1..j-1]$  ordinato, e  $j$  vale  $n+1$ .

---

<sup>1</sup>**ATTENZIONE:** verranno usati array con indici che partono da 1.

## 2 Lezione del 02/03/2018

### 2.1 Modello dei costi

**Assunzione** Tutte le istruzioni richiedono un tempo costante.  
Rivediamo l'algoritmo:

INSERTION-SORT( $A$ )

```
1   $n = A.length$ 
2  for  $j = 2$  to  $n$  // il primo elemento è già ordinato
3       $key = A[j]$  //  $A[1..j-1]$  ordinato
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i+1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i+1] = key$ 
```

Diamo il nome  $c_0$  alla chiamata del metodo, `InsertionSort(A)`; A ogni riga numerata, diamo il nome  $c_1, c_2, \dots, c_8$ <sup>1</sup>.

Vediamo il *costo* di ogni istruzione:

$$c_0 \rightarrow 1$$

$$c_1 \rightarrow 1$$

$$c_2 \rightarrow n$$

$$c_3 \rightarrow (n - 1)$$

$$c_4 \rightarrow (n - 1)$$

$$c_5 \rightarrow \sum_{j=2}^n t_j + 1$$

$$c_6, c_7 \rightarrow \sum_{j=2}^n t_j$$

$$c_8 \rightarrow (n - 1)$$

---

<sup>1</sup>( $c_1$  corrisponde alla riga 1,  $c_2$  alla riga 2 e così via).



## 2.2 Complessità di IS

$$T^{IS}(n) = c_0 + c_1 + c_2n + (c_3 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n (t_j + 1) + (c_6 + c_7) \sum_{j=2}^n t_j$$

$t_j$  dipende, oltre che da  $n$ , dall'istanza dell'array che stiamo considerando. È chiaro che questo calcolo non dà indicazioni precise sull'effettiva complessità dell'algoritmo.

Andiamo ad analizzare i 3 possibili casi:

- a) Caso migliore (2.2.1)
- b) Caso peggiore (2.2.2)
- c) Caso medio (2.2.3)

### 2.2.1 Caso migliore

→  $A$  ordinato  $\Rightarrow t_j = 0 \forall j$

La **complessità** diventa:

$$T_{min}^{IS}(n) = c_0 + c_1 + c_2n + (c_3 + c_4 + c_5 + c_8)(n - 1) = an + b \approx n$$

Ossia, si comporta come  $n$ . Il *caso migliore* **non** è interessante, visto che è improbabile si presenti.

### 2.2.2 Caso peggiore

→  $A$  ordinato in senso inverso  $\Rightarrow \forall j \ t_j = j - 1$

La **complessità** diventa:

$$T_{max}^{IS}(n) = c_0 + c_1 + c_2n + (c_3 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n j + (c_6 + c_7) \sum_{j=2}^n (j - 1)$$

Per valutare il costo di  $\sum_{j=2}^n j$  e di  $\sum_{j=2}^n (j - 1)$ , usiamo la **somma di Gauss**:

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

Otteniamo:

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

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

Per finire, ricalcoliamo  $T_{max}^{IS}(n)$

$$T_{max}^{IS}(n) = a'n^2 + b'n + c' \approx n^2$$

### 2.2.3 Caso medio

Il caso medio è *difficile da calcolare*, e in una considerevole parte dei casi, coincide con il caso peggiore.

Comunque, l'idea è la seguente:

$$\frac{\sum_{\text{perm. di input}} T^{IS}(p)}{n!} \approx n^2 \quad \text{posso pensare che } t_j \cong \frac{j-1}{2}$$

## 2.3 Divide et Impera

Un algoritmo di sorting *divide et impera* si può suddividere in 3 fasi:

**divide** divide il problema dato in sottoproblemi più piccoli;

**impera** risolve i sottoproblemi:

- ricorsivamente;
- la soluzione è nota (e.g. array con un elemento);

**combina** compone le soluzioni dei sottoproblemi in una soluzione del problema originale.

## 2.4 Merge Sort

**Merge Sort**<sup>1</sup> è un esempio di algoritmo *divide et impera*. Andiamo ad analizzarlo.

---

<sup>1</sup>Si consiglia di dare uno sguardo all'algoritmo anche da altre fonti, poichè presentarlo graficamente in L<sup>A</sup>T<sub>E</sub>X, come è stato visto a lezione, non è facile.

**Astrazione** Consideriamo il seguente array A.

5	2	4	7	1	2	3	6
---	---	---	---	---	---	---	---

Lo divido a metà, ottenendo due parti separate.

5	2	4	7
---	---	---	---

1	2	3	6
---	---	---	---

Consideriamo il primo, ossia A[1..4] (A originale). Divido anche questo a metà.

5	2
---	---

4	7
---	---

Divido nuovamente a metà, ottenendo:

5
---

2
---

5 e 2 sono due blocchi già ordinati. Scelgo il minore tra i due e lo metto in prima posizione, mentre l'altro in seconda posizione, ottenendo un blocco composto da 2 e 5.

Riprendo con il blocco composto da 4 e 7. Lo divido in due blocchi da un elemento. Faccio lo stesso procedimento fatto per 2 e 5: metto in prima posizione 4 e in seconda posizione 7. La situazione è la seguente:

2	5
---	---

4	7
---	---

So che i blocchi ottenuti contengono elementi ordinati. Con questa assunzione, posso ragionare nel seguente modo: considero il primo elemento dei due blocchi (2 e 4 in questo caso) e metto in prima posizione il minore tra i due. Ora considero il successivo elemento del blocco che è stato scelto e lo stesso elemento dell'altro blocco, e inserisco nell'array l'elemento minore. Continuo fino ad ottenere un blocco ordinato.

2	4	5	7
---	---	---	---

Faccio lo stesso procedimento con la parte di array originale A[5..8], ottenendo

2	4	5	7
---	---	---	---

1	2	3	6
---	---	---	---

A questo punto, i blocchi da 4 contengono elementi tra loro ordinati. Faccio lo stesso ragionamento usato per comporli, per ottenere l'array originale ordinato. Considero<sup>1</sup>:

---

<sup>1</sup>Questo procedimento è stato applicato anche ai passaggi precedenti; qui è spiegato più rigorosamente.

- $L[1..4] = A[1..4]$ : indice  $i = 1$  per scorrerlo;
- $R[1..4] = A[5..8]$ : indice  $j = 1$  per scorrerlo;

Valuto  $L[i]$  e  $R[j]$ .

- Se  $L[i] \leq R[j]$ , inserisco  $L[i]$  e incremento  $i$ .
- Altrimenti, inserisco  $R[j]$  e incremento  $j$ .
- Itero finchè entrambi gli indici non sono out of bounds.

**Pseudocodice** Segue lo pseudocodice del Merge Sort.

MERGE-SORT( $A, p, r$ )

```

1  if  $p < r$ 
2       $q = \lfloor \frac{p+r}{2} \rfloor$  // arrotondato per difetto
3      MERGE-SORT( $A, p, q$ ) // ordina  $A[p..q]$ 
4      MERGE-SORT( $A, q+1, r$ ) // ordina  $A[q+1..r]$ 
5      MERGE( $A, p, q, r$ ) // "Merge" dei due sotto-array

```

MERGE( $A, p, q, r$ )

```

1   $n1 = q - p + 1$  // gli indici partono da 1
2   $n2 = r - q$ 
   // L sotto-array sx, R sotto-array dx
3  for  $i = 1$  to  $n1$ 
4       $L[i] = A[p + i - 1]$ 
5  for  $j = 1$  to  $n2$ 
6       $R[j] = A[q + j]$ 
7   $L[n1 + 1] = R[n2 + 1] = \infty$ 
8   $i = j = 1$ 
9  for  $k = p$  to  $r$ 
10     if  $L[i] \leq R[j]$ 
11          $A[k] = L[i]$ 
12          $i = i + 1$ 
13     else //  $L[i] > R[j]$ 
14          $A[k] = R[j]$ 
15          $j = j + 1$ 

```

#### 2.4.1 Invarianti e correttezza

**L** e **R** contengono rispettivamente  $A[p..q]$  e  $A[q+1..r]$ . L'indice  $k$  scorre **A**. Il sotto-array  $A[p..k-1]$  è ordinato, e contiene  $L[1..i-1]$  e  $R[1..j-1]$ .

$$\begin{aligned} A[p..k-1] &\leq L[i..n1], R[j..n2] \\ &\quad \downarrow \\ A[p..k-1] &= A[p..r+1-1] \implies A[p..r] \text{ ordinato} \end{aligned}$$

**Dimostrazione per induzione su r-p**

$\Rightarrow$  Se  $r - p == 0$  (oppure  $-1$ ) abbiamo al più un elemento  $\implies$  array già ordinato.

$\Rightarrow$  Se  $r - p > 0$ , vale

$$\#elem(A[p..q]), \#elem(A[q+1..r]) < \#elem(A[p..r])$$

Per ipotesi induttiva:

- `Merge-sort(A, p, q)` ordina  $A[p..q]$ ;
  - `Merge-sort(A, q+1, r)` ordina  $A[q+1..r]$ ;
- Per correttezza di `Merge()`, dopo la sua chiamata ottengo  $A[p..r]$  ordinato.

## 3 Lezione del 07/03/2018

### 3.1 Approfondimento sull'induzione

#### 3.1.1 Induzione ordinaria

Proprietà  $P(n)$ , e.g.  $P(n) =$  “Se  $n$  è pari,  $n + 1$  è dispari” oppure “tutti i grafi con  $n$  nodi ...”.

Per dimostrare che  $P(n)$  vale per ogni  $n$

- $P(0)$ : **caso base**;
- assumo vera  $P(n) \rightarrow$  dimostro  $P(n+1)$ , allora  $P(n)$  è vera per ogni  $n$ .

#### 3.1.2 Induzione completa

- $[P(0)]$  (non necessaria, è un'istanza del passo successivo);
- dimostro  $P(m) \forall m < n \rightarrow$  vale  $P(n) \forall n$ .

### 3.2 Complessità di Merge Sort

$n = \#$ elementi da ordinare<sup>1</sup>

Merge(A,p,q,r)

**inizializzazione:**  $a'n + b'$ ;

**ciclo:**  $a'n + b'$ ;

Sommandoli, ottengo una complessità all'incirca di:

$$T^{merge}(n) = an + b$$

Nel dettaglio:

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(n_1) + T^{MS}(n_2) + T^{merge}(n) & \text{altrimenti} \end{cases}$$

$\Downarrow$

---

<sup>1</sup>Il simbolo  $\#$  verrà usato per indicare la cardinalità di un insieme.

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(n_1) + T^{MS}(n_2) + an + b & \text{altrimenti} \end{cases}$$

con

$$n_1 = \lfloor \frac{n}{2} \rfloor$$

$$n_2 = \lceil \frac{n}{2} \rceil$$

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(\lfloor \frac{n}{2} \rfloor) + T^{MS}(\lceil \frac{n}{2} \rceil) + an + b & \text{altrimenti} \end{cases}$$

$$\begin{array}{c} T^{MS}(n) \\ an + b \end{array}$$

$$\begin{array}{cc} T^{MS}(n_1) & T^{MS}(n_2) \\ an_1 + b & an_2 + b \end{array}$$

$$\begin{array}{cccc} T^{MS}(n_{11}) & T^{MS}(n_{12}) & T^{MS}(n_{21}) & T^{MS}(n_{22}) \\ an_{11} + b & an_{12} + b & an_{21} + b & an_{22} + b \end{array}$$

...

$$\begin{array}{cccccc} c_0 & c_0 & \dots & \dots & \dots & c_0 & c_0 \end{array}$$

Otteniamo  $c_0$  ripetuto  $n$  volte all'ultimo livello dell'albero. L'altezza dell'albero è circa  $\log_2 n$ . Vediamo nel dettaglio la complessità nelle varie iterazioni.

$$\mathbf{i = 0} \quad an + b$$

$$\mathbf{i = 1} \quad a(n_1 + n_2) + 2b \approx an + 2b$$

$$\mathbf{i = 2} \quad a(n_{11} + n_{12} + n_{21} + n_{22}) + 4b \approx an + 4b$$

...

$$\mathbf{i = h} \quad c_0 n$$

Poniamo  $n = 2^h$ . Abbiamo

$$\begin{aligned}
 T^{MS}(n) &= \sum_{i=0}^{h-1} (an + 2^i b) + c_0 n \\
 &= an h + b \sum_{i=0}^{h-1} 2^i && (h = \log_2 n) \\
 &= an \log_2 n + b 2^h - b + c_0 n && (2^h = n) \\
 &= an \log_2 n + (b + c_0)n - b \\
 T^{MS}(n) &= an \log_2 n + b''n + c'' \approx n \log_2 n
 \end{aligned}$$

### 3.3 Confronto tra IS e MS

$$\begin{aligned}
 T^{IS}(n) &= a'n^2 + b'n + c' \\
 T^{MS}(n) &= a''n \log_2 n + b''n + c''
 \end{aligned}$$

Posso calcolare il limite del rapporto:

$$\lim_{n \rightarrow +\infty} \frac{T^{MS}(n)}{T^{IS}(n)} = \lim_{n \rightarrow +\infty} \frac{a''n \log_2 n + b''n + c''}{a'n^2 + b'n + c'} = 0$$

Per definizione

$$\begin{aligned}
 \forall \varepsilon > 0 \quad \exists n_0 : \forall n \geq n_0 \quad \frac{T^{MS}(n)}{T^{IS}(n)} < \varepsilon \\
 \Downarrow \\
 T^{MS}(n) < \varepsilon T^{IS}(n) = \frac{T^{IS}}{m} \quad \left( \text{Ponendo, ad esempio, } \varepsilon = \frac{1}{m} \right)
 \end{aligned}$$

Detto a parole, c'è un certo  $n$  oltre il quale, ad esempio, **Merge Sort** su un *Commodore 64* esegue più velocemente di un **Insertion Sort** su una macchina moderna. Possiamo vedere una comparazione tra i due algoritmi nella seguente tabella.

$n$	$T^{IS}(n) = n^2$	$T^{MS}(n) = n \log n$
10	0.1ns	0.033ns
1000	1ms	10μs
$10^6$	17 minuti	20ms
$10^9$	70 anni	30s



## 4 Lezione dell'08/03/2018

### 4.1 Notazione asintotica

Il **tempo di esecuzione** è difficile da calcolare, come visto nella sezione 2.2. Il modo in cui è stato calcolato è pieno di dettagli “inutili”.

Rivediamo le complessità di Insertion Sort e Merge Sort:

$$\begin{aligned}T^{IS} &= an^2 + bn + c \\ T^{MS} &= an \log_2 n + bn + c\end{aligned}$$

A noi interessa calcolare  $T(n)$  per  $n$  “grande”. Non consideriamo le costanti moltiplicative, che sono non fondamentali. Ecco una lista di possibili complessità ordinate in senso decrescente (le prime due categorie appartengono alla classe degli *NP problems*, ossia non trattabili):

- $3^n$
- $2^n$
- $n^k$
- $n^2$
- $n \log n$
- $n$
- $\log n$
- 1

Prendiamo in esame due funzioni:  $f(n)$ ,  $g(n)$ :

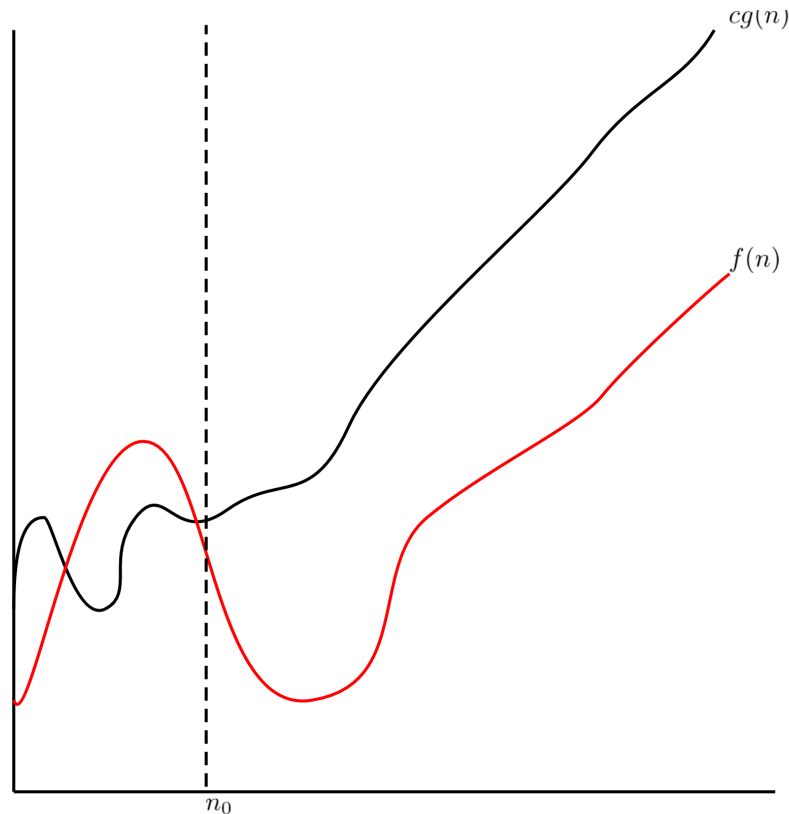
$$f, g : \mathbb{R}^+ \rightarrow \mathbb{R}^+$$

- $f(n)$  è la funzione in esame della complessità del nostro problema P;
- $g(n)$  è la funzione che, moltiplicata per un'opportuna costante  $c_i$ , dopo un certo  $n$ , fa da limite superiore o inferiore per ogni punto di  $f(n)$ .

#### 4.1.1 Limite asintotico superiore

Data  $g(n)$ , indichiamo con  $O(g(n))$  il *limite asintotico superiore*, definito come segue:

$$O(g(n)) = \{f(n) \mid \exists c > 0 \quad \exists n_0 (\in \mathbb{N}) \mid \forall n \geq n_0 \Rightarrow (0 \leq) f(n) \leq c \cdot g(n)\}$$



#### Esempi

◦  $f_1(n) = 2n^2 + 5n + 3 = O(g(n^2))$  ? Sì.

Deve valere  $f_1(n) < cn^2 \quad \exists c > 0, n \geq n_0$

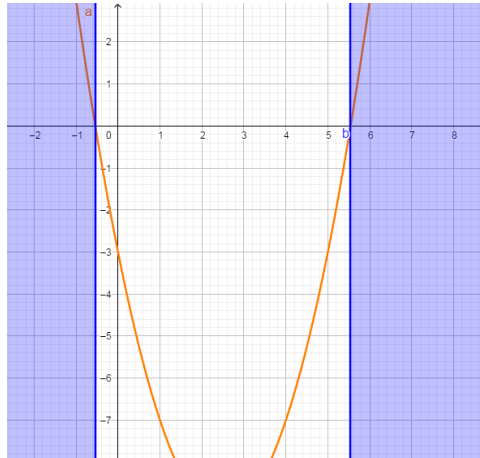
Ipotizziamo  $c = 3$

$$2n^2 + 5n + 3 \leq 3n^2$$

$$n^2 - 5n - 3 \geq 0$$

$$\frac{5 \pm \sqrt{2 \cdot 5 + 12}}{2} = \frac{5 \pm \sqrt{37}}{2} \cong 5.54$$

(Non considero la soluzione  
negativa, poiché siamo in  $\mathbb{R}^+$ )



Prendo  $c = 3$  e  $n_0 = 6$ . Vale dunque:

$$f_1(n) \leq cn^2 \quad \forall n \geq n_0$$

◦  $f_1(n) = O(g(n^3))$  ? Sì.

$$c = 3$$

$$n_0 = 6 \quad \forall n \geq n_0$$

$$f_1(n) \leq cn^2 \leq cn^3$$

◦  $f_2(n) = 2 + \sin(n) = O(1)$  ? Sì.

$$-1 \leq \sin(n) \leq 1$$

$$1 \leq f_2(n) \leq 3$$

Vale la seguente

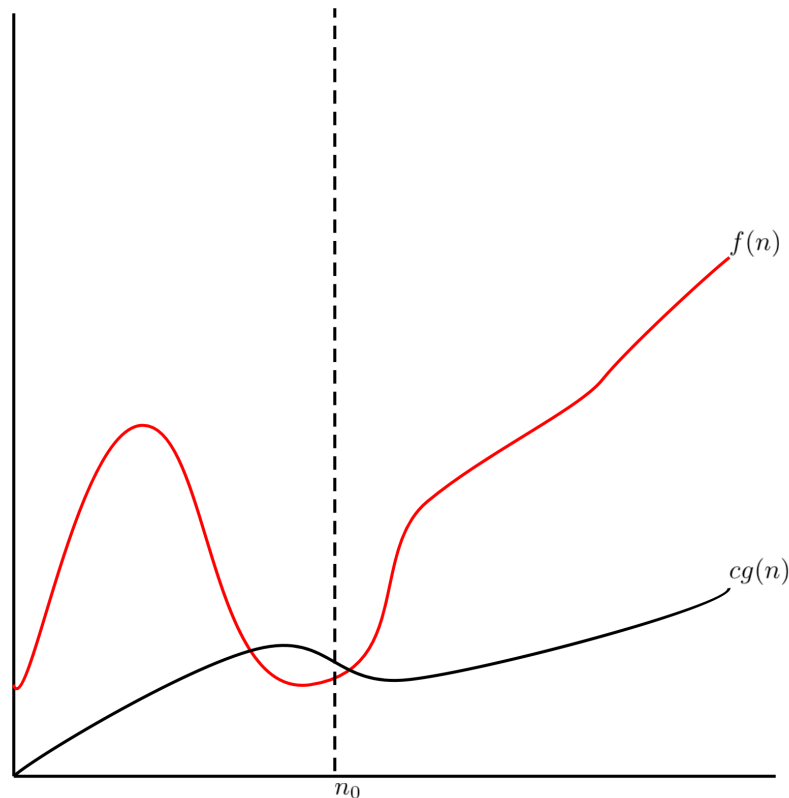
$$\exists c > 0 \quad \exists n_0 : n \geq n_0 \Rightarrow f_2(n) \leq c \cdot 1$$

ok per  $c = 3$ ,  $n_0 = 0$

### 4.1.2 Limite asintotico inferiore

Data  $g(n)$ , indichiamo con  $\Omega(g(n))$  il *limite asintotico inferiore*, definito come segue:

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0 \quad \exists n_0 (\in \mathbb{N}) \mid \forall n \geq n_0 \Rightarrow c \cdot g(n) \leq f(n)\}$$



### Esempi

- $f_1(n) = 2n^2 + 5n + 3 = \Omega(g(n^2))$  ? Sì.

Deve valere:

$$\exists c > 0 \quad \exists n_0 : \forall n \geq n_0 \Rightarrow cn^2 \leq 2n + 5n + 3$$

Basta porre  $c = 1$ ,  $n_0 = 0$ .

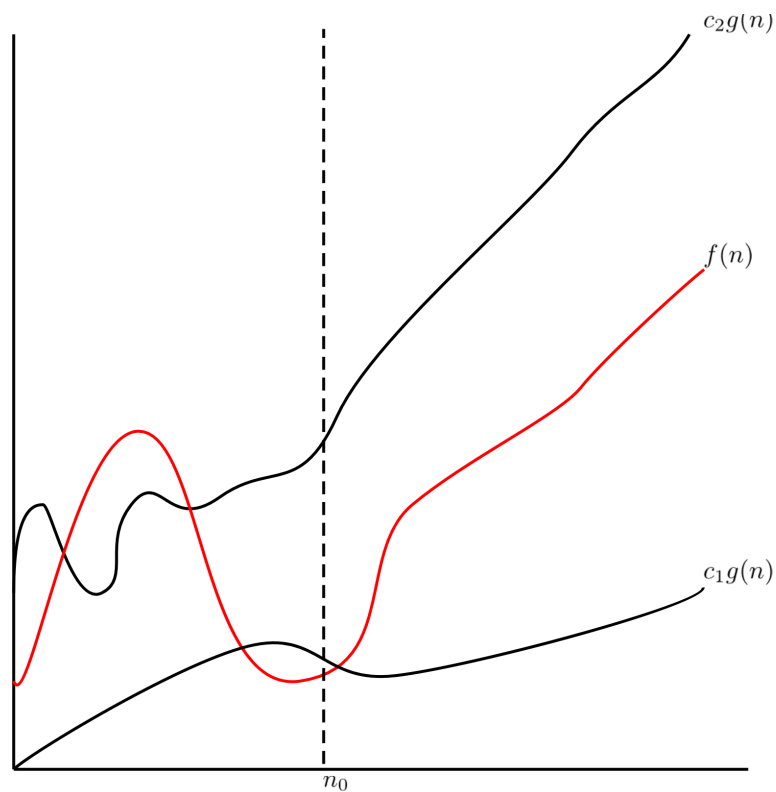
- $f_2(n) = 2 + \sin(n) = \Omega(1)$  ? Sì.

$$1 \leq f_2(n) \leq 3 \quad c = 1, \quad n_0 = 0$$

### 4.1.3 Limite asintotico stretto

Data  $g(n)$ , indichiamo con  $\Theta(g(n))$  il *limite asintotico stretto*, definito come segue:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0 \quad \exists n_0 (\in \mathbb{N}) \mid \forall n \geq n_0 \\ \Rightarrow c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$



### Esempi

$$\begin{aligned} f_1(n) &= 2n^2 + 5n + 3 = \Theta(n^2) \\ c_1 &= 1 \quad c_2 = 3 \quad n_0 = 6 \\ f_2(n) &= 2 + \sin(n) = \Theta(1) \\ c_1 &= 1 \quad c_2 = 3 \quad n_0 = 0 \end{aligned}$$

$$\begin{aligned} f_1(n) &\neq \Theta(n^3) \\ f_1(n) &= O(n^3) \\ f_1(n) &\neq \Omega(n^3) \end{aligned}$$

$$\Downarrow \\ \frac{f_1(n)}{n^3} \rightarrow 0$$

## 4.2 Metodo del limite

$$f(n), g(n) > 0 \quad \forall n$$

Se  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)}$  esiste, allora:

1. Se  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = k > 0$  allora  $f(n) = \Theta(g(n))$ .

$$\begin{aligned} \text{Infatti } \forall \varepsilon > 0 \exists n_0 : \forall n \geq n_0 &\Rightarrow \left| \frac{f(n)}{g(n)} - k \right| \leq \varepsilon \\ &\Rightarrow -\varepsilon \leq \frac{f(n)}{g(n)} - k \leq \varepsilon \end{aligned}$$

$$k - \varepsilon \leq \frac{f(n)}{g(n)} \leq k + \varepsilon$$

$$(k - \varepsilon)g(n) \leq f(n) \leq (k + \varepsilon)g(n) \quad \text{per } 0 < \varepsilon < k$$

2. Se  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$  allora  $f(n) = O(g(n))$  e  $f(n) \neq \Omega(g(n))$ .
3. Se  $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = \infty$  allora  $f(n) = \Omega(g(n))$  e  $f(n) \neq O(g(n))$ .

## 4.3 Alcune proprietà generali

- $f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = \Theta(n^k)$
- $h \neq k \quad \Theta(n^h) \neq \Theta(n^k)$
- $a \neq b \quad \Theta(a^k) \neq \Theta(b^k)$
- $h \neq k \quad \Theta(a^{n+h}) = \Theta(a^{n+k})$
- $a \neq b \quad \Theta(\log_a n) = \Theta(\log_b n)$

In generale

$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq \dots$$

Rivediamo **Insertion Sort** con le notazioni asintotiche:

$$T^{IS}(n) = O(n^2) \quad T_{max}^{IS}(n) = \Theta(n^2)$$

Vale anche la proprietà seguente:

$$\begin{aligned} 2n^k + \Theta(n^{k-1}) &= O(n^k) (\subseteq \Theta(n^k)) \\ &= \Theta(n^k) \quad \forall k > 0 \end{aligned}$$

## 5 Lezione del 09/03/2018

### 5.1 Complessità di un problema

Dato un problema<sup>1</sup>  $P$  ci sono (possono esserci) algoritmi che risolvono  $P$ . La **complessità** di  $P$  è la complessità dell'algoritmo di complessità più bassa che lo risolve.

**Limite superiore per complessità di  $P$**  Se  $A$  è un algoritmo per  $P$  con complessità  $f(n)$ , allora  $P$  è  $O(f(n))$ .

Vediamo un paio di esempi:

- Insertion Sort algoritmo di ordinamento  $O(n^2)$ ;
- Merge Sort algoritmo di ordinamento  $O(n \log n)$ .

**Limite inferiore per complessità di  $P$**  Se ogni algoritmo che risolve  $P$  ha complessità  $\Omega(f(n))$  allora  $P$  è  $\Omega(f(n))$

$$\implies \text{ se } P \text{ è } O(f(n)) \text{ e } \Omega(f(n)) \Rightarrow P \text{ è } \Theta(f(n))$$

### 5.2 Esempio: limite inferiore per l'ordinamento basato su scambi

**Def (inversione)** Dato  $A[1..n]$ , una *inversione* è una coppia  $(i, j)$  con  $i, j \in [1, n]$  con  $i < j$  e  $A[i] > A[j]$ .

Operazione disponibile:  $A[k] \leftrightarrow A[k+1]$  (scambio tra gli elementi in posizione  $k$  e  $k+1$ ).

$$\begin{aligned} \#inv(A) &= \text{numero di inversioni di } A \\ &= \left| \{(i, j) \mid 1 \leq i < j \leq n \wedge A[i] > A[j]\} \right| \end{aligned}$$

1.  $A$  è ordinato sse  $\#inv(A) = 0$ ;
2.  $A$  è ordinato in senso inverso sse

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

Ossia,  $\#inv(A)$  è massimizzato.

---

<sup>1</sup>Relazione/funzione INPUT  $\rightarrow$  OUTPUT

Vediamo cosa succede alle coppie  $(i, j)$  e a  $\#inv(A)$  nel caso avvenga uno scambio  $A[k] \leftrightarrow A[k+1]$ .

- $i, j \neq k$  e  $i, j \neq k+1 \implies (i, j)$  è inversione prima sse è inversione dopo;
- $i = k, j = k+1$

$$\implies \begin{cases} A[k] < A[k+1] & +1 \text{ inversione} \\ A[k] = A[k+1] & \#inv(A) \text{ non cambia} \\ A[k] > A[k+1] & -1 \text{ inversione} \end{cases}$$

- $i = k$  oppure  $i = k+1, j > k+1 \implies (k, j)$  è inversione prima sse  $(k+1, j)$  è inversione dopo;
- $j = k$  oppure  $j = k+1, i < k$ , analogo al caso precedente.

Per concludere, possiamo dire che l'operazione  $A[k] \leftrightarrow A[k+1]$  riduce  $\#inv(A)$  al massimo di 1.

$$\implies \text{qualsunque algoritmo di ordinamento è } \Omega\left(\frac{n(n-1)}{2}\right) = \Omega(n^2)$$

Insertion Sort è “quasi” basato su scambi  $\Rightarrow$  è  $O(n^2) \Rightarrow$  è  $\Theta(n^2)$

### 5.3 Soluzione di ricorrenze

Abbiamo visto per Merge Sort la complessità nel modo seguente:

MERGE-SORT( $A, p, r$ )

```

1  if  $p < r$ 
2       $q = \lfloor \frac{p+r}{2} \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q+1, r$ )
5      MERGE( $A, p, q, r$ ) // complessità  $an + b$ 
```

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(\lfloor \frac{n}{2} \rfloor) + T^{MS}(\lceil \frac{n}{2} \rceil) + an + b & \text{se } n > 1 \end{cases}$$

È stato tuttavia un approccio non molto preciso. Ci sono due metodi per risolvere precisamente i problemi di ricorrenza:

- Metodo di sostituzione (5.3.1);
- Master Theorem (7.1).



### 5.3.1 Metodo di sostituzione

Dato una ricorrenza, si può provare a “indovinare” la soluzione, oppure si può sviluppare l'*albero delle ricorrenze*:

- *radice*: chiamata di cui vogliamo la complessità;
- per ogni nodo:
  - costo della parte non ricorsiva;
  - un figlio per ogni chiamata.

#### Esempio

$$T(n) = \begin{cases} 4 & \text{se } n = 1 \\ 2T(\frac{n}{2}) + 6n & \text{se } n > 1 \end{cases}$$

In generale, si può benissimo trascurare il caso base per poter ottenere espressioni meno verbose, in questo caso otterremmo:

$$T(n) = 2T(\frac{n}{2}) + 6n$$

Per questa volta, facciamo il procedimento per intero. Proviamo a “indovinare” la soluzione<sup>1</sup>. Assomiglia a **Merge Sort**, quindi ipotizziamo abbia una complessità con un andamento simile

$$T(n) = an \log n + bn + c$$

Facciamo la prova induttiva.

$$\begin{array}{ll} (n = 1) & T(1) = 4 \\ & = a \cdot 1 \cdot \log 1 + b \cdot 1 + c & (\log 1 = 0) \\ & = b + c & \text{ok se } b + c = 4 \\ (n > 1) & T(n) = 2T(\frac{n}{2}) + 6n \end{array}$$

---

<sup>1</sup>In classe, è stato visto anche un esempio con un albero. Ho scelto di ometterlo per la poca praticità nel rappresentarlo in L<sup>A</sup>T<sub>E</sub>X.

Per ipotesi induttiva

$$T\left(\frac{n}{2}\right) = a\frac{n}{2} \cdot \log \frac{n}{2} + b\frac{n}{2} + c$$

Calcolo ora  $T(n)$

$$\begin{aligned} T(n) &= an \log_2 \frac{n}{2} + bn + 2c + 6n = \\ &= an \log_2 n - an \log_2 2 + bn + 6n + 2c = \quad (\log_2 2 = 1) \\ &= an \log_2 n + n(b + 6 - a) + 2c = \\ &= an \log_2 n + bn + c \\ &\quad \Downarrow \end{aligned}$$

$$b + 6 - a = b \Rightarrow a = 6$$

$$2c = c \Rightarrow c = 0$$

$$b + c = 4 \Rightarrow b = 4$$

$$\begin{aligned} T(n) &= an \log n + bn + c \\ &= 6n \log n + 4n \end{aligned}$$

## 6 Lezione del 14/03/2018

### Esercizio (importante)

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 6n \\ &= 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n) \\ \text{vale } \exists c > 0 \exists n_0 : \forall n \geq n_0 \Rightarrow \Theta(n) &\leq cn \end{aligned}$$

Voglio dimostrare che

1.  $T(n) = O(n \log n)$
  2.  $T(n) = \Omega(n \log n)$
1.  $T(n) = O(n \log n)$

significa che  $\exists d > 0 \exists n_1 \in \mathbb{N} \mid T(n) \leq dn \log n \quad \forall n \geq n_1$

Dimostro per induzione  $T(n) \leq dn \log n \quad \forall n \geq n_1$ .

Ometto il caso base, poiché non è molto interessante (mi basterebbe aumentare ulteriormente  $d$  per avere un valore accettabile).

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn && \text{ip. induttiva } T\left(\frac{n}{2}\right) = d\frac{n}{2} \log \frac{n}{2} \\ &\leq 2 \cdot \frac{n}{2} d \log \frac{n}{2} + cn && \left(\log \frac{n}{2} = \log n - \log 2\right) \\ &= dn \log n - dn \log 2 + cn \\ &= dn \log n - n(d \log 2 - c) \leq dn \log n \\ &\Rightarrow -n(d \log 2 - c) \leq 0 \\ n(d \log 2 - c) &\geq 0 \\ d \log 2 - c &\geq 0 \\ d &\geq \frac{c}{\log 2} \end{aligned}$$

2.  $T(n) = \Omega(n \log n)$  è analoga.

$$\exists \delta > 0 : \forall n > n_0 \Rightarrow T(n) \geq \delta n \log n$$

Ho l'ipotesi induttiva  $T(\frac{n}{2}) \geq \delta \frac{n}{2} \log \frac{n}{2}$

$$\begin{aligned} T(n) &\geq 2\delta \frac{n}{2} \log \frac{n}{2} + cn = \\ &= \delta n \log n - \delta n \log 2 + cn = \\ &= \delta n \log n + n(c - \delta \log 2) \geq \delta n \log n \\ &\text{Deve valere } c - \delta \log 2 \geq 0 \\ &\Rightarrow 0 < \delta \leq \frac{c}{\log 2} \end{aligned}$$

**Esercizio**  $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \Theta(n)$  ( $\Theta(n) \leq c \cdot n$ )

Ipotizzo un andamento simile a Merge Sort:  $\Theta(n \log n)$ . Dimostro:

1.  $T(n) = O(n \log n)$

2.  $T(n) = \Omega(n \log n)$

1.  $T(n) = \Omega(n \log n)$

$$\exists d > 0 : \forall n > n_0 \Rightarrow T(n) \leq dn \log n$$

Ometto il caso base. L'ipotesi induttiva è la seguente:

$$T(n) \leq d \frac{n}{3} \log \frac{n}{3} + d \frac{2n}{3} \log \frac{2n}{3} + cn$$

Procedo con i calcoli ...

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \\ &\leq d \frac{n}{3} \log \frac{n}{3} + \frac{2n}{3} \log \frac{2n}{3} = \\ &= \frac{dn}{3} \left( \log n - \log 3 \right) + d \frac{2n}{3} \left( \log n - \log \frac{2}{3} \right) + cn = \\ &= dn \log n - \frac{dn}{3} \left( \log 3 - 2 \log \frac{2}{3} \right) + cn = \\ &= dn \log n - \frac{dn}{3} \left( \log 3 - \log \frac{4}{9} \right) + cn = \\ &= dn \log n - n \left( \frac{d}{3} \log \frac{27}{4} - c \right) \leq dn \log n \\ &\quad \frac{d}{3} \log \frac{27}{4} - c \geq 0 \\ &\Rightarrow d \geq \frac{3c}{\log \frac{27}{4}} \quad \left( \log \frac{27}{4} > 1 \text{ poiché } \arg > 1 \right) \end{aligned}$$

**2.**  $T(n) = \Omega(n \log n)$  è analoga

$$\exists \delta > 0 : \forall n > n_0 \Rightarrow T(n) \geq \delta n \log n$$

L'ipotesi induttiva è la seguente:

$$T(n) \geq \delta \frac{n}{3} \log \frac{n}{3} + \delta \frac{2n}{3} \log \frac{2n}{3} + cn$$

Calcoli ...

$$\begin{aligned} T(n) &\geq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \\ &\geq \delta \frac{n}{3} \log \frac{n}{3} + \frac{2n}{3} \log \frac{2n}{3} = \\ &= \delta \frac{n}{3} \left( \log n - \log 3 \right) + \delta \frac{2n}{3} \left( \log n - \log \frac{2}{3} \right) + cn = \\ &= \delta n \log n + \frac{\delta n}{3} \left( -\log 3 + 2 \log \frac{2}{3} \right) + cn = \\ &= \delta n \log n + \frac{\delta n}{3} \left( -\log 3 + \log \frac{4}{9} \right) + cn = \\ &= \delta n \log n + n \left( -\frac{\delta}{3} \log \frac{27}{4} + c \right) \geq \delta n \log n \\ &\quad - \frac{\delta}{3} \log \frac{27}{4} + c \geq 0 \\ &\Rightarrow 0 < \delta \leq \frac{3c}{\log \frac{27}{4}} \end{aligned}$$

## 7 Lezione del 15/03/2018

### 7.1 Master Theorem

Dato un problema con **size**  $n$ , vogliamo dividerlo in  $a$  sottoproblemi con **size**  $\frac{n}{b}$ . Otteniamo la seguente ricorrenza (ricordiamo che il caso base è omesso per semplicità):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

con  $a \geq 1$ ,  $b > 1$ , allora possiamo confrontare

- $f(n)$ ;
- $n^{\log_b a}$ .

Tre possibili casi:

1. Se  $f(n) = O(n^{\log_b a - \varepsilon})$  per qualche  $\varepsilon > 0$ , allora

$$T(n) = \Theta(n^{\log_b a})$$

2. Se  $f(n) = \Theta(n^{\log_b a})$  allora

$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

3. Se  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  per qualche  $\varepsilon > 0$ , e vale la *regolarità*

$$\exists 0 < k < 1 \text{ tale che } a \cdot f\left(\frac{n}{b}\right) \leq k \cdot f(n)$$

allora

$$T(n) = \Theta(f(n))$$

**Breve “dimostrazione” sul perchè  $n^{\log_b a}$**

$$T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2f\left(\frac{n}{b^2}\right) + \dots + a^{\log_b n}f\left(\frac{n}{b^{\log_b n}}\right) + c \cdot a^{\log_b n}$$

$$a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}$$

$$\text{Nota bene: } af\left(\frac{n}{b}\right) \leq k \cdot f(n) \text{ con } k < 1$$

Vediamo ora i casi in cui sarà possibile finire, e le conclusioni legate ad essi.

A)

$$\lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a}} = l (> 0) \neq \infty$$

$$\text{Caso 2} \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

B)

$$\lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a}} = 0$$

Potrei essere nel *Caso 1*  $\Rightarrow$  se  $\lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a - \varepsilon}} = l \neq \infty$  ( $\varepsilon > 0$ )

$$\Rightarrow T(n) = \Theta(n^{\log_b a})$$

C)

$$\lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a}} = \infty \quad \& \quad \exists \varepsilon > 0 : \lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a + \varepsilon}} = l \neq 0$$

$$\& \text{Regolarit\`a} \Rightarrow \text{Caso 3: } T(n) = \Theta(f(n))$$

### 7.1.1 Esercizi (Master Theorem)

- $T^{MS} = 2T\left(\frac{n}{2}\right) + a'n + b'$

Abbiamo (rispetto alla forma  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ )

$$a = 2, \quad b = 2$$

$$f(n) = a'n + b' \quad n^{\log_2 2} = n$$

È chiaro che le due funzioni hanno lo stesso andamento (di ordine  $\Theta(n)$ ):

$$a'n + b' = \Theta(n)$$

$$\text{Caso 2} \Rightarrow T(n) = \Theta\left(n^{\log_2 2} \log n\right) = \Theta(n \log n)$$

- $T(n) = 5T\left(\frac{n}{2}\right) + 2n^2 + n \log n$

Abbiamo (rispetto alla forma  $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ )

$$a = 5, \quad b = 2$$

$$f(n) = n^2 + n \log n \quad n^{\log_2 5} \quad (\log_2 5 > 2)$$

$$0 < \varepsilon < \log_2 5 - 2 \Rightarrow \lim_{n \rightarrow \infty} \frac{2n^2 + n \log n}{n^{\log_2 5 - \varepsilon}} = 0 \Rightarrow f(n) = O(n^{\log_2 5})$$

$$\text{Caso 1} \Rightarrow T(n) = \Theta(n^{\log_2 5})$$

- $T(n) = 5T(\frac{n}{2}) + n^3$  per esercizio.
- $T(n) = 5T(\frac{n}{2}) + n^3 \log n$

Abbiamo

$$\begin{aligned} a &= 5, \quad b = 2 \\ f(n) &= n^3 \log n \quad n^{\log_2 5} \quad (\log_2 5 < 3) \\ 0 < \varepsilon < 3 - \log_2 5 &\Rightarrow \lim_{n \rightarrow \infty} \frac{n^3 \log n}{n^{\log_2 5 + \varepsilon}} = \infty \end{aligned}$$

Possibile caso 3. *Regolarità?*

$$\begin{aligned} af\left(\frac{n}{b}\right) &\leq kf(n) \quad \text{per } 0 < k < 1 \text{ opportuno} \\ 5\left(\frac{n}{2}\right)^3 \log \frac{n}{2} &= \frac{5}{8}n^3 \log \frac{n}{2} \leq \frac{5}{8}n^3 \log n \leq kn^3 \log n \quad \text{per } 0 < k \leq \frac{5}{8} < 1 \\ &\Downarrow \\ \text{Caso 3: } T(n) &= \Theta(f(n)) = \Theta(n^3 \log n) \end{aligned}$$

- $T(n) = 27T(\frac{n}{3}) + n^3 \log n$

$$\begin{aligned} f(n) &= n^3 \log n \quad n^{\log_3 27} \quad (\log_3 27 = 3) \\ \lim_{n \rightarrow \infty} \frac{n^3 \log n}{n^{3+\varepsilon}} &= +\infty \quad \forall \varepsilon > 0, \text{ non possiamo dimostrare } 3 \\ &\Rightarrow \text{Non siamo in nessun caso del Master Theorem.} \end{aligned}$$

Anche valutando la *regolarità*, ricadiamo in un assurdo. Dobbiamo dimostrare che  $af(\frac{n}{b}) < kf(n)$  per qualche  $k > 0$

$$\begin{aligned} 27\left(\frac{n}{3}\right)^3 \log \frac{n}{3} &= n^3(\log n - \log 3) \not\leq kn^3 \log n \quad \text{per nessun } k > 0 \\ \text{Infatti } \frac{(\log n - \log 3)n^3}{n^3 \log n} &\rightarrow 1 \end{aligned}$$

(Posso usare il Metodo di Sostituzione)

$$T(n) = 27T\left(\frac{n}{3}\right) + n^3 \log n$$

Costruiamo l'albero delle ricorrenze:

• *radice*: costo  $n^3 \log n$ ;



· ogni nodo ha 27 figli.

- ◇ i 27 figli del primo livello hanno costo  $(\frac{n}{3})^3 \log \frac{n}{3}$ ;
- ◇ i  $27^2$  figli del secondo livello hanno costo  $(\frac{n}{9})^3 \log \frac{n}{9}$ ;
- ◇ ...
- ◇ le  $27^n$  foglie terminali hanno costo  $O(1)$ .

$$\begin{aligned} T(n) &= \sum_{j=0}^{\log_3 n} n^3 \log \frac{n}{3^j} = n^3 \sum_{j=0}^{\log_3 n} (\log n - j \log 3) + cn = \\ &= n^3 (\log n)^2 - n^3 \log 3 \sum_{j=0}^{\log_3 n} j + cn \quad \left( \sum_{j=0}^{\log_3 n} j \cong (\log_3 n)^2 \right) \end{aligned}$$

$$T(n) = 27T\left(\frac{n}{3}\right) + n^3 \log n$$

$$T(n) = \Theta(n^3 (\log n)^2) \quad \text{ipotesi ricavata}$$

Devo dimostrare che valgono le seguenti condizioni:

$$T(n) = \Omega(n^3 (\log n)^2)$$

$$1. T(n) = O(n^3 (\log n)^2)$$

$$T(n) \leq c \cdot n^3 (n^3 (\log n)^2) \quad c > 0$$

$$T(n) = 27T\left(\frac{n}{3}\right) + n^3 \log n$$

$$\begin{aligned} &\left( \text{ipotesi induttiva } T\left(\frac{n}{3}\right) \leq c \cdot \left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 \right) \\ &\leq 27c \left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 + n^3 \log n = \\ &= \frac{27cn^3}{27} (\log n - \log 3)^2 + n^3 \log n = \\ &= cn^3 \left( (\log n)^2 - 2 \log 3 \log n + (\log 3)^2 \right) + n^3 \log n = \\ &= cn^3 (\log n)^2 - n^3 \left( \log n (2c \log 3 - 1) - c(\log 3)^2 \right) \\ &\leq cn^3 (\log n)^2 \end{aligned}$$

Per un  $n$  abbastanza grande, vale la disuguaglianza con un opportuno valore di  $c$ :

$$c > \frac{1}{2 \log 3}$$

$$2. \ T(n) = \Omega(n^3(\log n)^2)$$

$$\begin{aligned} \exists d > 0 : T(n) &\geq dn^3(\log n)^2 \\ &\geq 27\left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 + n^3 \log n \\ &= \dots = dn^3(\log n)^2 - n^3 \left( \log n(2d \log 3 - 1) - d(\log 3)^2 \right) \\ &\geq dn^3(\log n)^2 \end{aligned}$$

Per un  $n$  abbastanza grande, vale la disuguaglianza con un opportuno valore di  $d$ :

$$2d \log 3 - 1 < 0 \quad \text{ok per } 0 < d < \frac{1}{2 \log 3}$$

## 8 Lezione del 21/03/2018

**Ordinamento** Finora abbiamo visto due algoritmi di ordinamento, in cui avevamo le seguenti premesse:

IN:  $a_1 \dots a_n$ ;

OUT: permutazione  $a'_1 \dots a'_n$  ordinata.

In particolare, abbiamo concluso che:

- **Insertion Sort**:  $O(n^2)$ , basato su scambi;
- **Merge Sort**:  $\Theta(n \log n)$ , ma con un costo in termini di *memoria*.

### Memoria

- **Insertion Sort**:

$input + 1$  variabile  $\Rightarrow$  spazio *costante*  $\Theta(1)$  (detto “in loco”)

- **Merge Sort**: spazio con costo lineare.

$$\begin{aligned} S_{MS}(n) &= \max \left\{ S\left(\left\lfloor \frac{n}{2} \right\rfloor\right), S\left(\left\lceil \frac{n}{2} \right\rceil\right), \Theta(n) \right\} \\ &= \Theta(n) \end{aligned}$$

### 8.1 Heapsort

L'**Heapsort**<sup>1</sup> è un algoritmo di ordinamento basato su una struttura chiamata **heap**, che prende le caratteristiche positive di **Insertion Sort** e **Merge Sort**:

- in “loco” (spazio  $\Theta(1)$ );
- complessità  $\Theta(n \log n)$ .

**Cos'è un heap?** Un **heap** è una struttura dati basata sugli alberi che soddisfa la “proprietà di heap”: se A è un genitore di B, allora la chiave di A è ordinata rispetto alla chiave di B conformemente alla relazione d'ordine applicata all'intero heap.

Seguono alcune definizioni.

---

<sup>1</sup>Anche qui, si consiglia di dare un occhio ad altre fonti. In classe, sono stati viste molte rappresentazioni grafiche degli heap, e, come già detto, in  $\text{\LaTeX}$  non è per me facile rappresentarli.

**Altezza:** è la distanza dalla radice alla foglia più distante;

**Albero completo:** è un albero di altezza  $h$  con  $\sum_{i=0}^h 2^i - 1$  nodi;

**Albero quasi completo:** è un albero completo a tutti i livelli eccetto l'ultimo, in cui possono mancare delle foglie.

Gli heap verranno rappresentati in array monodimensionali, nel modo descritto di seguito:

$$\forall i > 0$$

- $A[i]$  è il nodo genitore;
- $A[2i]$  è il figlio sx del nodo  $A[i]$ ;
- $A[2i+1]$  è il figlio dx.

Inoltre, ogni array  $A$  sarà dinamico, e avrà:

- $A.length$  potenziale spazio, capacità massima dell'array;
- $A.heapsize$  celle effettive dell'array.

Vediamo alcune funzioni di utilità che verranno usate.

LEFT( $i$ )

// restituisce il figlio sx del nodo  $i$

1 **return**  $2 * i$

RIGHT( $i$ )

// restituisce il figlio dx del nodo  $i$

1 **return**  $2 * i + 1$

PARENT( $i$ )

// restituisce il genitore del nodo  $i$

1 **return**  $\lfloor i/2 \rfloor$

### 8.1.1 Max Heap

*Max Heap* è uno heap che soddisfa la seguente proprietà:

$$\begin{aligned} &\forall \text{ nodo } A[i], \\ &A[i] \geq \text{discendenti} \\ &\Downarrow \\ &A[i] \geq A[\text{Left}(i)], A[\text{Right}(i)] \end{aligned}$$

**Osservazioni**

- Uno heap con un solo elemento è un *Max Heap*.
- Dati due Max Heap  $T_1$  e  $T_2$  e un nodo  $N$ , possiamo “combinarli” in uno heap con  $N$  come radice,  $T_1$  come *left* e  $T_2$  come *right*.

Ecco ora una procedura che, dato un nodo  $i$ , trasforma in un Max Heap il sotto-albero eradicato in esso (con radice  $i$ ).

MAXHEAPIFY( $A, i$ )

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if ( $l \leq A.\text{heapsize}$ ) and ( $A[l] > A[i]$ )
4  else
5       $max = i$ 
6  if ( $max \neq i$ )
7       $A[i] \leftrightarrow A[max]$ 
8      MAXHEAPIFY( $A, max$ )
```

L'algoritmo ha un costo di  $O(h)$ , con  $h$  altezza del sotto-albero radicato in  $i$ , con

$$O(h) \cong O(\log n) \quad (\text{Omessa la dimostrazione})$$

Ora vogliamo scrivere una procedura che costruisce un *Max Heap* da un array qualunque.

Quali sono i nodi foglia?

- Se  $i \geq \lfloor \frac{n}{2} \rfloor + 1$

$$\begin{aligned} 2i &= 2\left(\frac{n}{2} + 1\right) \geq n + 2 - 1 = n + 1 \\ &\Rightarrow i \text{ foglia} \end{aligned}$$

- Se  $i \leq \lfloor \frac{n}{2} \rfloor$

$$\begin{aligned} 2i &= 2\lfloor \frac{n}{2} \rfloor \leq n \\ &\Rightarrow i \text{ non foglia} \end{aligned}$$

BUILDMAXHEAP( $A$ )

```
1   $A.\text{heapsize} = A.\text{length}$ 
2  for  $i = \lfloor A.\text{length}/2 \rfloor$  down to 1
3      MAXHEAPIFY( $A, i$ )
```

---

# Appendices

## A Raccolta algoritmi

### A.1 Insertion Sort

Per approfondire, vedi la sezione 1.4

INSERTION-SORT( $A$ )

```
1   $n = A.length$ 
2  for  $j = 2$  to  $n$  // il primo elemento è già ordinato
3       $key = A[j]$  //  $A[1..j-1]$  ordinato
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i+1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i+1] = key$ 
```

### A.2 Merge Sort

Vedi la sezione 2.4

MERGE-SORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \lfloor \frac{p+r}{2} \rfloor$  // arrotondato per difetto
3      MERGE-SORT( $A, p, q$ ) // ordina  $A[p..q]$ 
4      MERGE-SORT( $A, q+1, r$ ) // ordina  $A[q+1..r]$ 
5      MERGE( $A, p, q, r$ ) // "Merge" dei due sotto-array
```

MERGE( $A, p, q, r$ )

```

1   $n1 = q - p + 1$  // gli indici partono da 1
2   $n2 = r - q$ 
   // L sotto-array sx, R sotto-array dx
3  for  $i = 1$  to  $n1$ 
4       $L[i] = A[p + i - 1]$ 
5  for  $j = 1$  to  $n2$ 
6       $R[j] = A[q + j]$ 
7   $L[n1 + 1] = R[n2 + 1] = \infty$ 
8   $i = j = 1$ 
9  for  $k = p$  to  $r$ 
10     if  $L[i] \leq R[j]$ 
11          $A[k] = L[i]$ 
12          $i = i + 1$ 
13     else //  $L[i] > R[j]$ 
14          $A[k] = R[j]$ 
15          $j = j + 1$ 

```

### A.3 Insertion Sort ricorsivo

INSERTION-SORT( $A, j$ )

```

1  if  $j < 1$ 
2      INSERTION-SORT( $A, j - 1$ ) // ordina  $A[1..j-1]$ 
3      INSERT( $A, j$ ) // inserisce  $A[j]$  in modo ordinato in  $A$ 

```

INSERT( $A, j$ )

```

   // Precondizione:  $A[1..j-1]$  è ordinato
1  if ( $j > 1$ ) and ( $A[j] < A[j - 1]$ )
2       $A[j] \leftrightarrow A[j - 1]$  // scambia le celle  $j$  e  $j-1$ 
   // se le celle sono state scambiate, ordina
   // il nuovo sottoarray  $A[1..j-1]$ 
3      INSERT( $A, j - 1$ )

```

#### A.3.1 Correttezza di Insertion-Sort( $A, j$ )

Procediamo per induzione:

( $j \leq 1$ ) Caso base. Array già ordinato, non faccio nulla  $\Rightarrow$  ok;

( $j > 1$ ) Per ipotesi induttiva, la chiamata INSERTION-SORT( $A, j-1$ ) ordina  $A[1..j-1]$ . Assumendo la correttezza di INSERT( $A, j-1$ ), esso “inserisce”  $A[j]$   $\Rightarrow$  produce  $A[1..j]$  ordinato.

### A.3.2 Correttezza di `Insert(A, j)`

Anche qui, dimostrazione per induzione:

( $j = 1$ ) Caso base. `A[1]` da inserire nell'array vuoto. Non fa nulla  $\Rightarrow$  ok;

( $j > 1$ ) Due sottocasi:

- $A[j] \geq A[j-1]$ : non faccio nulla, `A[1..j]` già ordinato;
- $A[j] < A[j-1]$ : scambio le chiavi delle due celle. Il nuovo `A[j]` sarà sicuramente maggiore di qualsiasi altro elemento che lo precede, poiché, per preconditione di `Insert`, `A[1..j-1]` era ordinato, e dato che valeva  $A[j-1] \geq A[j]$ , il nuovo `A[j]` (che è il precedente `A[j-1]`) sarà sicuramente l'elemento con il valore più alto. Dopodichè, chiamo `Insert(A, j-1)` per ordinare la cella `A[j-1]`.

## A.4 CheckDup

Algoritmo che verifica la presenza di duplicati in `A[p..r]` e, solo se non ci sono, ordina l'array.

Se `A[p..q]` e `A[q+1..r]` ordinati e privi di duplicati:

- Se `A[p..r]` non contiene duplicati, ordina e restituisce **false**;
- altrimenti, restituisce **true**.

`CHECK-DUP(A, p, r)`

```
1  if  $p < r$ 
2       $q = \lfloor \frac{p+r}{2} \rfloor$  // arrotondato per difetto
3      return CHECK-DUP(A, p, q)
4          or CHECK-DUP(A, q + 1, r)
5          or DMERGE(A, p, q, r)
```



DMERGE( $A, p, q, r$ )

```

1   $n1 = q - p + 1$  // gli indici partono da 1
2   $n2 = r - q$ 
   // L sotto-array sx, R sotto-array dx
3  for  $i = 1$  to  $n1$ 
4       $L[i] = A[p + i - 1]$ 
5  for  $j = 1$  to  $n2$ 
6       $R[j] = A[q + j]$ 
7   $L[n1 + 1] = R[n2 + 1] = \infty$ 
8   $i = j = 1$ 
9  while ( $k \leq p$ ) and ( $L[i] \neq R[j]$ )
10     if  $L[i] < R[j]$ 
11          $A[k] = L[i]$ 
12          $i = i + 1$ 
13     else //  $L[i] > R[j]$ 
14          $A[k] = R[j]$ 
15          $j = j + 1$ 
16      $k = k + 1$ 
17 return  $k \leq r$ 

```

#### A.4.1 Correttezza di DMerge(A,p,q,r)

- $A[p..k-1]$  è ordinato, contiene  $L[1..i-1] \cup R[1..j-1]$ ;
- $A[p..k-1] < L[1..n1], R[1..n2]$ .

### A.5 SumKey

Dato  $A[1..n]$  e  $key$  intera, Sum( $A, key$ ) restituisce:

- **true** se  $\exists i, j \in [1, n] : key = A[i] + A[j]$ ;
- **false** altrimenti.

Vediamo una prima versione, non efficiente, dell'algoritmo. Ha complessità  $O(n^2)$ .

SUMB( $A, key$ )

```

1   $n = A.length$ 
2   $i = j = 1$ 
3  while ( $i \leq n$ ) and ( $A[i] + A[j] \neq key$ )
4      if  $j = n$ 
5           $i = i + 1$ 
6      else
7           $j = j + 1$ 
8  return  $i \leq n$ 

```

Ecco ora una versione più efficiente, che però richiede un **sorting** preventivo, che quindi causa *side effect*. Si assume un algoritmo di sorting con complessità  $O(n \log n)$ . Con questa premessa, la ricerca della coppia di valori ha complessità  $O(n)$  nel caso peggiore. Nel complesso, vale quindi:

$$O(n \log n + n) = O(n \log n)$$

SUM( $A, key$ )

```

1   $n = A.length$ 
2  SORT( $A$ ) // complessità  $O(n \log n)$ 
3   $i = 1, j = n$ 
4  while ( $i \leq j$ ) and ( $A[i] + A[j] \neq key$ )
5      if  $A[i] + A[j] < key$ 
6           $i = i + 1$ 
7      else
8           $j = j - 1$ 
9  return  $i \leq j$ 

```

#### A.5.1 Correttezza di Sum( $A, key$ )

Valgono i seguenti invarianti:

- (1)  $\forall h \in [1, i - 1], \forall k \in [h, n] \Rightarrow A[h] + A[k] \neq key$
- (2)  $\forall k \in [j + 1, n], \forall h \in [1, k] \Rightarrow A[k] + A[h] \neq key$

Supponiamo di trovarci in  $A[i] + A[j] < key$

→ incremento  $i$ ;

(1) **non** cambia;

(2) (vogliamo dimostrare)  $\forall k \in [i, n] \quad A[i] + A[k] \neq key$ .

Distinguiamo 2 casi.

## A. Raccolta algoritmi

---

· Siccome vale  $A[k] \leq A[j]$ , allora

$$A[i] + A[k] \leq A[i] + A[j] > key$$

·  $k \in [j + 1, n]$  quindi

$$A[i] + A[k] \neq key \text{ per (2)}$$

Se esco perché  $i > j$ , **non** c'è una soluzione poiché

$$(1) + (2) \Rightarrow \forall h \leq k \quad A[h] + A[k] \neq key$$

Presetiamo ora una terza soluzione, che però richiede un costo in memoria direttamente proporzionale al valore *max* (che chiameremo *top*) dell'array considerato, poiché richiede di allocare un array *V* di booleani di dimensione dipendente da *top*, in cui il valore **A[i]** corrisponde alla cella **V[A[i]]**. Assumiamo

$$A[i] \geq 0 \quad \forall i \in [i, n], \quad key \leq top$$

$$V[v] = \text{true} \text{ sse } \exists i : A[i] = v$$

SUMV(*A*, *key*)

```

1  V[0..key] ← FALSE // Θ(key) = O(top) = O(1)
2  i = 1
3  found = FALSE
4  while (i ≤ n) and not found
5      if A[i] ≤ key
6          V[A[i]] = TRUE
7          found = V[key - A[i]]
8      i = i + 1
9  return found
```

Complessità:

- $O(n)$  se *top* costante;
- $O(n \cdot key)$  altrimenti.

## B Esercizi

### B.1 Ricorrenze

- $T(n) = aT(n-1) + b \quad a, b > 1$ 
  - *radice*: costo  $b$ ;
  - la radice ha  $a$  figli di costo  $b$ ;
  - ...
  - foglie terminali  $O(1)$ .

Esplicitando il caso base della ricorrenza otteniamo:

$$T(n) = \begin{cases} c & n = 0 \\ aT(n-1) + b & n > 0 \end{cases}$$

$$\begin{aligned} T(n) &= b + ab + a^2b + \dots + a^{n-1}b + a^nc \\ &= b \sum_{j=0}^{n-1} a^j + a^nc \quad (\text{dimostrare per induzione}) \end{aligned}$$

$$(a = 1) \quad T(n) = nb + c = \Theta(n)$$

$$(a < 1) \quad T(n) = \frac{1-a^n}{1-a} \cdot b + a^nc = \Theta(1)$$

$$(\text{valgono } \frac{1-a^n}{1-a} \leq \frac{1}{1-a}, \quad a^nc < c)$$

$$(a > 1) \quad T(n) = \frac{a^n-1}{a-1}b + a^nc = \Theta(a^n)$$