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

24 marzo 2018

Indice

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.

$$\text{Input} \rightarrow \textbf{Algoritmo} \rightarrow \text{Output}$$

Dato un problema, ci sono tanti algoritmi per risolverlo.

e.g.[1] Ordinamento dei numeri di una `Rubrica`. L'idea è quella di trovare tutte le permutazioni di ogni numero.

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

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).

[1] 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 sicurezzza. 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.

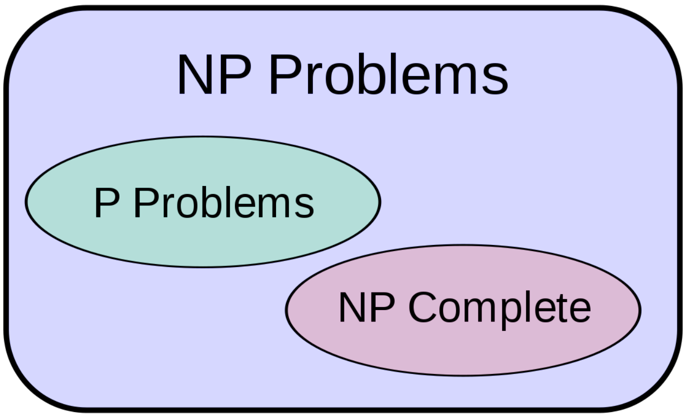


Figura 1: Complessità T(n).

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 inverto con l'elemento alla sua sinistra, come segue

2	5	8	4	7

Key: $\boxed{8}$

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

2	5	8	4	7

Key: $\boxed{4}$

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: $\boxed{7}$

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

Algorimo Passiamo ora all'implementazione dell'algoritmo, con uno pseudocodice similare a `Python`[1]

`Input`: $A[1,..,n]$, $A.length$.

$$\text{È noto che: } A[i] \le key < A[i+1]$$

Pseudocodice Segue lo pseudocodice dell'`Insertion Sort`.

```
INSERTION-SORT(A)
n = A.length
for j = 2 to n // il primo elemento è già ordinato
    key = A[j] // A[1..j-1] ordinato
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    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]` $\le$ `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`.

[1] **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 $c_1, c_2, .., c_8$ [1].

Vediamo il *costo* di ogni istruzione:

$$\boldsymbol{c_0} \rightarrow 1$$

$$\boldsymbol{c_1} \rightarrow 1$$

$$\boldsymbol{c_2} \rightarrow n$$

$$\boldsymbol{c_3} \rightarrow (n-1)$$

$$\boldsymbol{c_4} \rightarrow (n-1)$$

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

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

$$\boldsymbol{c_8} \rightarrow (n-1)$$

[1] (c_1 corrisponde alla riga 1, c_2 alla riga 2 e così via).

2.2 Complessità di IS

$$T^{IS}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} (t_j + 1) + (c_6 + c_7) \sum_{j=2}^{n} t_j$$

$\boldsymbol{t_j}$ dipende, oltre che da n, dall'istanza dell'array che stiamo considerando. È chiaro che questo calcolo non da 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

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

La **complessità** diventa:

$$T^{IS}_{min}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_5 + c_8)(n-1) = an + b \approx n$$

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

2.2.2 Caso peggiore

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

La **complessità** diventa:

$$T^{IS}_{max}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} j + (c_6 + c_7) \sum_{j=2}^{n} (j-1)$$

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 $T^{IS}_{max}(n)$

$$T^{IS}_{max}(n) = a'n^2 + b'n + c' \approx n^2$$

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:

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

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[1] è un esempio di algoritmo *divide et impera*. Andiamo ad analizzarlo.

[1] Si consiglia di dare uno sguardo all'algoritmo anche da altre fonti, poichè presentarlo graficamente in LaTeX, 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[1]:

[1] 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)
if p < r
    q = ⌊(p+r)/2⌋ // arrotondato per difetto
    MERGE-SORT(A, p, q) // ordina A[p..q]
    MERGE-SORT(A, q + 1, r) // ordina A[q+1..r]
    MERGE(A, p, q, r) // "Merge" dei due sotto-array
```

```
MERGE(A, p, q, r)
n1 = q − p + 1 // gli indici partono da 1
n2 = r − q
// L sotto-array sx, R sotto-array dx
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = R[n2 + 1] = ∞
i = j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else // L[i] > R[j]
        A[k] = R[j]
        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]`.

$$A[p \mathinner{.\,.} k-1] \leq L[i \mathinner{.\,.} n1], R[j \mathinner{.\,.} n2]$$
$$\Downarrow$$
$$A[p \mathinner{.\,.} k-1] = A[p \mathinner{.\,.} r+1-1] \implies A[p \mathinner{.\,.} r] \text{ ordinato}$$

Dimostrazione per induzione su r-p

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

⇒ Se $r - p > 0$, vale

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

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 $P(n) \to$ dimostro $P(n+1)$, allora $P(n)$ è vera per ogni n.

3.1.2 Induzione completa

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

3.2 Complessità di Merge Sort

$n =$ #elementi da ordinare[1]

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:

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

$$\Downarrow$$

[1]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

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

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

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

$$T^{MS}(n)$$
$$an + b$$

$T^{MS}(n_1)$ $\quad$ $T^{MS}(n_2)$

$an_1 + b$ $\quad$ $an_2 + b$

$T^{MS}(n_{11})$ $\quad$ $T^{MS}(n_{12})$ $\quad$ $T^{MS}(n_{21})$ $\quad$ $T^{MS}(n_{22})$

$an_{11} + b$ $\quad$ $an_{12} + b$ $\quad$ $an_{21} + b$ $\quad$ $an_{22} + b$

$\dots$

c_0 $\quad$ c_0 $\quad$ $\dots$ $\quad$ $\dots$ $\quad$ $\dots$ $\quad$ c_0 $\quad$ c_0

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.

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

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

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

$\dots$

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

Poniamo $n = 2^h$. Abbiamo

$$\begin{aligned}
T^{MS}(n) &= \sum_{i=0}^{h-1}(an + 2^i b) + c_0 n \\
&= anh + b\sum_{i=0}^{h-1} 2^i && (h = \log_2 n) \\
&= an\log_2 n + b2^h - b + c_0 n && (2^h = n) \\
&= an\log_2 n + (b + c_0)n - b
\end{aligned}$$

$$T^{MS}(n) = an\log_2 n + b''n + c'' \approx n\log_2 n$$

3.3 Confronto tra IS e MS

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

Posso calcolare il limite del rapporto:

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

Per definizione

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

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	$1ms$	$10\mu s$
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:

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

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: $f(n)$, $g(n)$:

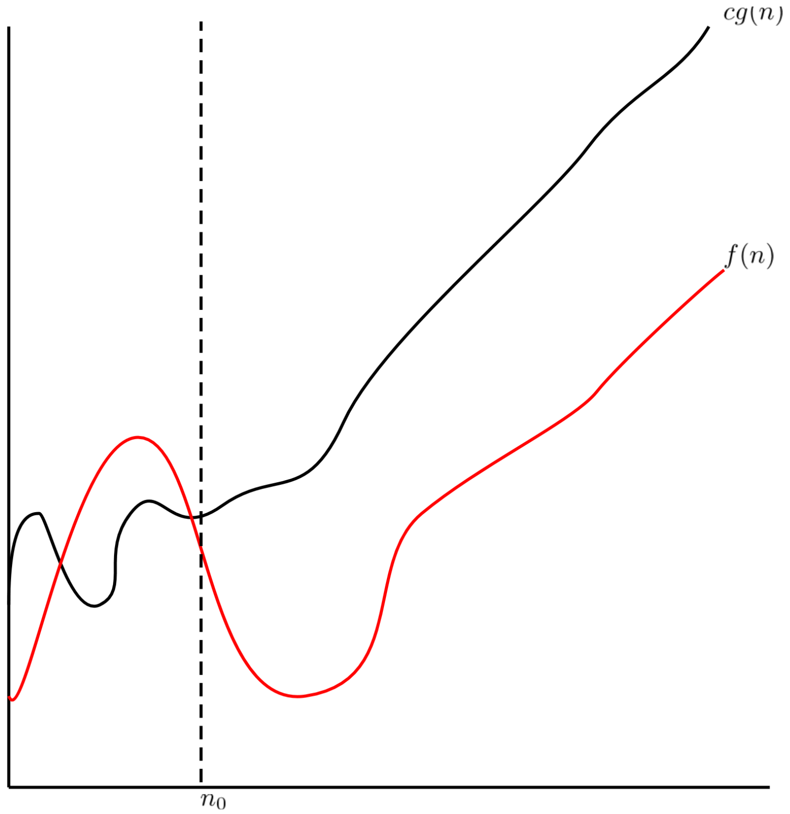
$$f, g : \mathbb{R}^+ \to \mathbb{R}^+$$

- $f(n)$ 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 $O\big(g(n)\big)$ il *limite asintotico superiore*, definito come segue:

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



Esempi

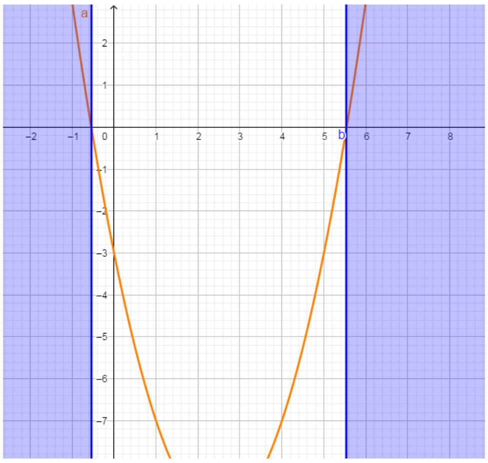
- $f_1(n) = 2n^2 + 5n + 3 = O\big(g(n^2)\big)$? Sì.

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

  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:

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

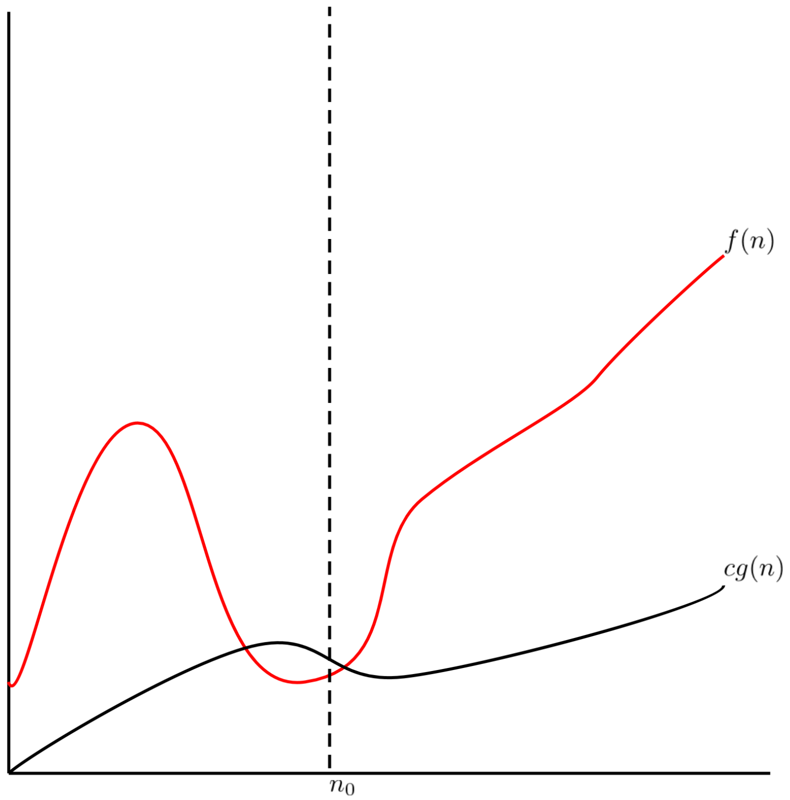
- $f_1(n) = O\big(g(n^3)\big)$? Sì.
  $c = 3$
  $n_0 = 6 \quad \forall n \geq n_0$
  $f_1(n) \leq cn^2 \leq cn^3$
- $f_2(n) = 2 + \sin(n) = O(1)$? Sì.
  $-1 \leq \sin(n) \leq 1$
  $1 \leq f_2(n) \leq 3$
  Vale la seguente

$$\exists c > 0 \quad \exists n_0 : n \geq n_0 \Rightarrow f_2(n) \leq c \cdot 1$$
$$\text{ok per } c = 3,\ n_0 = 0$$

4.1.2 Limite asintotico inferiore

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

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



Esempi

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

  Deve valere:

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

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

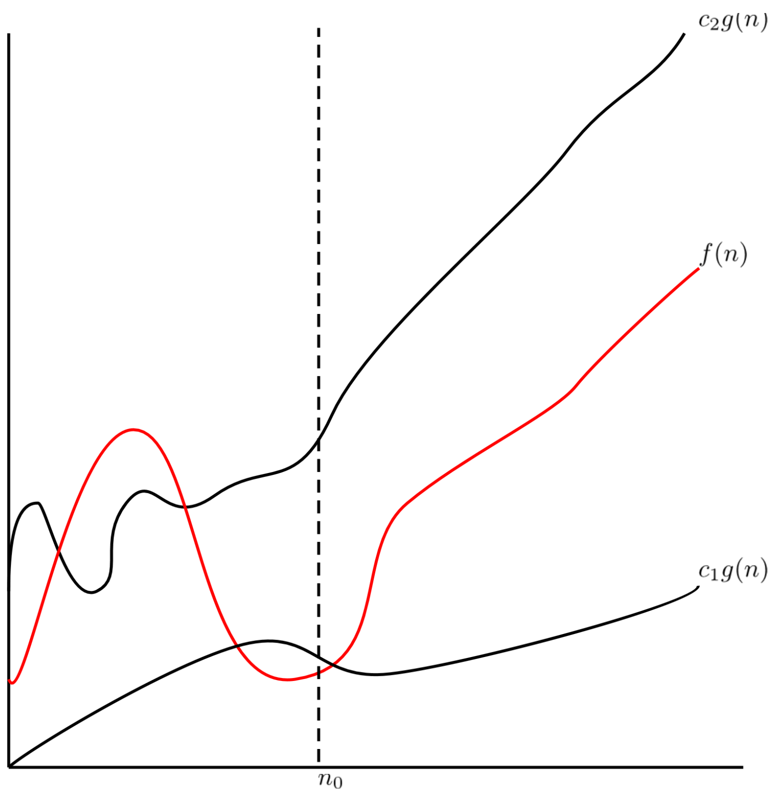
- $f_2(n) = 2 + \sin(n) = \Omega(1)$? Sì.

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

4.1.3 Limite asintotico stretto

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

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



Esempi

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

$$\begin{gathered} f_1(n) \neq \Theta(n^3) \\ f_1(n) = O(n^3) \\ f_1(n) \neq \Omega(n^3) \\ \Downarrow \\ \frac{f_1(n)}{n_3} \to 0 \end{gathered}$$

4.2 Metodo del limite

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

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

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

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

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

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

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

4.3 Alcune proprietà generali

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

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:

$$T^{IS}(n) = O(n^2) \qquad T^{IS}_{max}(n) = \Theta(n^2)$$

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[1] 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 $O\big(f(n)\big)$.

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 $\Omega\big(f(n)\big)$ allora P è $\Omega\big(f(n)\big)$

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

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 $k+1$).

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

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.

[1] 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.

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

`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)
if $p < r$
  $q = \lfloor \frac{(p+r)}{2} \rfloor$
  Merge-Sort(A, p, q)
  Merge-Sort$(A, q+1, r)$
  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[1]. Assomiglia a `Merge Sort`, quindi ipotizziamo abbia una complessità con un andamento simile

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

Facciamo la prova induttiva.

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

[1]In classe, è stato visto anche un esempio con un albero. Ho scelto di ometterlo per la poca praticità nel rappresentarlo in LaTeX.

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

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

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).

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

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

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

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

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 ...

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

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 ...

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

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
$$T(n) = \Theta\big(n^{\log_b a}\big)$$
2. Se $f(n) = \Theta(n^{\log_b a})$ allora
$$T(n) = \Theta\big(n^{\log_b a} \cdot \log n\big)$$
3. Se $f(n) = \Omega(n^{\log_b a} + \varepsilon)$ per qualche $\varepsilon > 0$, e vale la *regolarità*
$$\exists 0 < k < 1 \text{ tale che } a \cdot f(\frac{n}{b}) \leq k \cdot f(n)$$
allora
$$T(n) = \Theta\big(f(n)\big)$$

Breve "dimostrazione" sul perchè $n^{\log_b a}$

$$T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots + a^{\log_b n} f\left(\frac{n}{b^{\log_b n}}\right) + c \cdot a^{\log_b n}$$
$$a^{\log_b n} = \left(b^{\log_b a}\right)^{\log_b n} = \left(b^{\log_b n}\right)^{\log_b a} = n^{\log_b a}$$
$$\text{Nota bene: } af\left(\frac{n}{b}\right) \leq k \cdot f(n) \text{ con } k < 1$$

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

A)

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = l(>0) \neq \infty$$
$$\textit{Caso 2} \Rightarrow T(n) = \Theta\big(n^{\log_b a} \cdot \log n\big)$$

B)

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

$$\text{Potrei essere nel } \textit{Caso 1} \Rightarrow \text{se } \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a-\varepsilon}} = l \neq \infty \ (\varepsilon > 0)$$
$$\Rightarrow T(n) = \Theta\big(n^{\log_b a}\big)$$

C)

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = \infty \quad \& \ \exists\varepsilon > 0 : \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a+\varepsilon}} = l \neq 0$$
$$\& \ \textit{Regolarità} \Rightarrow \textit{Caso 3:} \quad T(n) = \Theta(f(n))$$

7.1.1 Esercizi (Master Theorem)

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

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

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

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

  $$a'n + b' = \Theta(n)$$
  $$\textit{Caso 2} \Rightarrow T(n) = \Theta\Big(n^{\log_2 2} \log n\Big) = \Theta(n \log n)$$

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

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

  $$a = 5, \ b = 2$$
  $$f(n) = n^2 + n\log n \qquad n^{\log_2 5} \quad (\log_2 5 > 2)$$
  $$0 < \varepsilon < \log_2 5 - 2 \Rightarrow \lim_{n\to\infty} \frac{2n^2 + n\log n}{n^{\log_2 5-\varepsilon}} = 0 \Rightarrow f(n) = O(n^{\log_2 5})$$
  $$\textit{Caso 1} \Rightarrow T(n) = \Theta\big(n^{\log_2 5}\big)$$

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

  Abbiamo

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

Possibile *caso 3*. *Regolarità*?

$$af\Big(\frac{n}{b}\Big) \le kf(n) \quad \text{per } 0 < k < 1 \text{ opportuno}$$
$$5\Big(\frac{n}{2}\Big)^3 \log\frac{n}{2} = \frac{5}{8}n^3 \log\frac{n}{2} \le \frac{5}{8}n^3 \log n \le kn^3 \log n \qquad \text{per } 0 < k \le \frac{5}{8} < 1$$
$$\Downarrow$$
$$\textit{Caso 3}\text{: } T(n) = \Theta\big(f(n)\big) = \Theta(n^3 \log n)$$

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

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

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

$$27\Big(\frac{n}{3}\Big)^3 \log\frac{n}{3} = n^3(\log n - \log 3) \not< kn^3 \log n \ \text{ per nessun } k > 0$$
$$\text{Infatti } \frac{(\log n - \log 3)n^3}{n^3 \log n} \to 1$$

(Posso usare il Metodo di Sostituzione)

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

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 $O(1)$.

$$T(n) = \sum_{j=0}^{\log_3 n} n^3 \log \frac{n}{3^j} = n^3 \sum_{j=0}^{\log_3 n} (\log n - j \log 3) + cn =$$

$$= n^3 (\log n)^2 - n^3 \log 3 \sum_{j=0}^{\log_3 n} j + cn \qquad \Big(\sum_{j=0}^{\log_3 n} j \cong (\log_3 n)^2 \Big)$$

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

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

Devo dimostrare che valgano le seguenti condizioni:

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

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

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

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

$$\Big(\text{ipotesi induttiva } T\Big(\frac{n}{3}\Big) \leq c \cdot \Big(\frac{n}{3}\Big)^3 \Big(\log \frac{n}{3}\Big)^2 \Big)$$

$$\leq 27c \Big(\frac{n}{3}\Big)^3 \Big(\log \frac{n}{3}\Big)^2 + n^3 \log n =$$

$$= \frac{\cancel{27} c n^3}{\cancel{27}} (\log n - \log 3)^2 + n^3 \log n =$$

$$= cn^3 \Big((\log n)^2 - 2 \log 3 \log n + (\log 3)^2 \Big) + n^3 \log n =$$

$$= cn^3 (\log n)^2 - n^3 \Big(\log n (2c \log 3 - 1) - c(\log 3)^2 \Big)$$

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

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

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

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

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

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

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

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`:

$$\textit{input}\ +\ 1 \text{ variabile} \Rightarrow \text{spazio } \textit{costante } \Theta(1) \text{ (detto “in loco”)}$$

- `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[1] 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.

[1] 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 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
    return 2 * i
```

```
RIGHT(i)
    // restituisce il figlio dx del nodo i
    return 2 * i + 1
```

```
PARENT(i)
    // restituisce il genitore del nodo i
    return ⌊i/2⌋
```

8.1.1 Max Heap

Max Heap è uno heap che soddisfa la seguente proprietà:

$$\forall \text{ nodo } A[i],$$
$$A[i] \geq \text{discendenti}$$
$$\Downarrow$$
$$A[i] \geq A[\mathit{Left(i)}],\ A[\mathit{Right(i)}]$$

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 radie i).

```
MaxHeapify(A, i)
l = Left(i)
r = Right(i)
if (l ≤ A.heapsize) and (A[l] > A[i])
else
      max = i
if (max ≠ i)
      A[i] ↔ A[max]
      MaxHeapify(A, max)
```

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

$$O(h) \cong O(\log n) \qquad \text{(omessa la dimostrazione)}$$

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

Quali sono i nodi foglia?

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

$$2i = 2\Big(\frac{n}{2} + 1\Big) \geq n + 2 - 1 = n + 1$$
$$\Rightarrow i \text{ foglia}$$

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

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

```
BuildMaxHeap(A)
A.heapsize = A.length
for i = ⌊A.length/2⌋ down to 1
      MaxHeapify(A, i)
```

L'algoritmo esegue $\frac{n}{2}$ volte `MaxHeapify`, che ha un costo di $O(n \log n)$, tuttavia questa stima è molto pessimistica.

Definiamo:

- h_T altezza del cammino più lungo dello heap;
- $h_T - 1$ di conseguenza è l'altezza dell'albero meno l'ultimo livello, che è generalmente incompleto.

$$
\begin{aligned}
n &= \left(2^{(h_T-1)+1} - 1\right) + 1 \\
&= 2^{h_T} \\
&h_T \leq \log n \\
&n \geq 2^{h_T}
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= \sum_{h=1}^{\lfloor \log n \rfloor} 2^{h_T - h} \cdot O(h) \\
&\qquad (2^{h_T-h} = \#\text{ chiamate a } \texttt{MaxHeapify} \text{ al livello } h) \\
&= \sum_{h=1}^{\lfloor \log n \rfloor} \frac{2^{h_T}}{2^h} O(h) \qquad (2^{h_T} = n) \\
&= O\Big(\Big(\sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}\Big) n\Big) = O(n) \qquad \Big(\sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \sum_{h=1}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2\Big)
\end{aligned}
$$

Passiamo ora all'algoritmo di ordinamento `Heapsort`. La radice di un *Max Heap* contiene il valore massimo. Quindi, la prima operazione, e quella su cui si basa `Heapsort`, consiste nel mettere la radice in ultima posizione.

Es. `A`: 9 8 7 5 7 4 0 4 3 6 1 2 — è un max heap.
$\Rightarrow$ 8 7 5 7 4 0 4 3 6 1 2 9 — ignoro l'ultimo elemento, chiamo `MaxHeapify` sulla radice e itero.

Poi chiama `MaxHeapify` sul resto dell'array per renderlo un *Max Heap*, e itera il procedimento sul nuovo array.

HeapSort(A)
BuildMaxHeap(A) // $O(n)$
for $i = A.length$ **down to** 2
    $A[1] \leftrightarrow A[i]$
    $A.heapsize = A.heapsize - 1$
    MaxHeapify($A, 1$) // $O(\log n)$

Costo? Il costo complessivo è di $O(n \log n)$.

8.2 Code con priorità

S insieme dinamico di oggetti.

x è l'indice, $x.key$ è il corrispondente valore relativo a quell'indice. Voglio poter eseguire le seguenti operazioni:

- `Insert(S, x)`
- `Max(S)`
- `ExtractMax(S)`
- `IncreaseKey(S, x, Δ)`
- `ChangeKey(S, x, Δ)`
- `Delete(S, x)`

Idea Uso un *Max Heap* (A).

Max(A)
if $A.heapsize = 0$
  error
else return $A[1]$

La procedura `Max(A)` ha complessità costante $\Theta(1)$.

ExtractMax(A)
$max = A[1]$
$A[1] = A[A.heapsize]$
$A.heapsize = A.heapsize - 1$
MaxHeapify($A, 1$) // ripristina le proprietà di *MaxHeap*
return max

La procedura `ExtractMax(A)` ha la stessa complessità di `MaxHeapify`: $O(\log n)$.

Per `Insert`, le cose diventano più delicate. L'idea è quella di inserire in coda ad A: in questo modo, l'unico elemento che potrebbe compromettere la proprietà di *Max Heap* è la cella di indice i (nel nostro caso, l'ultima). Deve vale la proprietà:

$$\text{Per ogni } j \neq i$$
$$A[j] \leq \text{antenati}$$

Non possiamo dire nulla su i. Va ristabilita la proprietà di *Max Heap*: per fare ciò usiamo la procedura `MaxHeapifyUp`.

MaxHeapifyUp(A, i)
if $(i > 1)$ **and** $(A[i] > A[\text{Parent}(i)])$
    $A[i] \leftrightarrow A[\text{Parent}(i)]$
    MaxHeapifyUp$(A, \text{Parent}(i))$

Correttezza di MaxHeapifyUp

Casi base

`(i = 1)` ok, non faccio nulla;

`(A[i]` $\leq$ `A[Parent(i)])` ok, la proprietà di *Max Heap* è mantenuta.

Induzione

`(A[i] > A[Parent(i)])` scambio le due celle. I discendenti (sotto-alberi) della nuova cella `A[i]` mantengono la proprietà di *Max Heap*.

Costo? $O(\log i)$, nel caso peggiore $O(\log n)$.

Ecco ora lo pseudocodice della funzione `Insert`.

Insert(A, x)
$A.\mathit{heapsize} = A.\mathit{heapsize} + 1$
$A[A.\mathit{heapsize}] = x$
MaxHeapifyUp$(A, A.\mathit{heapsize})$

`Insert` ha costo $O(\log n)$, lo stesso di `MaxHeapifyUp`.

IncreaseKey(A, i, δ)
// *Precondizione*: $\delta \geq 0$
$A[i] = A[i] + \delta$
MaxHeapifyUp(A, i)

`IncreaseKey` ha costo $O(\log n)$.

ChangeKey(A, i, δ)
$A[i] = A[i] + \delta$
if $\delta > 0$
    MaxHeapifyUp(A, i)
else // $\delta \leq 0$
    MaxHeapify(A, i)

`ChangeKey` è come `IncreaseKey`, ma può utilizzare valori di δ qualsiasi, ed è corretto per la seguente proprietà:

$$\begin{aligned}&\text{Se per ogni } j \neq i \quad A[j] \geq \text{discendenti}\\ &\Rightarrow \text{dopo } \texttt{MaxHeapify} \text{ ho un } \mathit{MaxHeap}\end{aligned}$$

```
DeleteKey(A, i)
old = A[i]
A[i] = A[A.heapsize]
A.heapsize = A.heapsize − 1
if old ≤ A[i]
      MaxHeapifyUp(A, i)
else
      MaxHeapify(A, i)
```

`DeleteKey` ha costo $O(\log n)$.

Appendices

A Raccolta algoritmi

A.1 Insertion Sort

Per approfondire, vedi la sezione 1.4

Insertion-Sort(A)

$n = A.length$
for $j = 2$ **to** n // il primo elemento è già ordinato
    $key = A[j]$ // $A[1\,..\,j-1]$ ordinato
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

A.2 Merge Sort

Vedi la sezione 2.4

Merge-Sort(A, p, r)

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

```
MERGE(A, p, q, r)
n1 = q − p + 1 // gli indici partono da 1
n2 = r − q
// L sotto-array sx, R sotto-array dx
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = R[n2 + 1] = ∞
i = j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else // L[i] > R[j]
        A[k] = R[j]
        j = j + 1
```

A.3 Insertion Sort ricorsivo

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

```
INSERT(A, j)
// Precondizione: A[1..j-1] è ordinato
if (j > 1) and (A[j] < A[j − 1])
    A[j] ↔ A[j − 1] // scambia le celle j e j-1
    // se le celle sono state scambiate, ordina
    // il nuovo sottoarray A[1..j-1]
    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 precondizione 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)

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

```
DMerge(A, p, q, r)
n1 = q − p + 1 // gli indici partono da 1
n2 = r − q
// L sotto-array sx, R sotto-array dx
for i = 1 to n1
      L[i] = A[p + i − 1]
for j = 1 to n2
      R[j] = A[q + j]
L[n1 + 1] = R[n2 + 1] = ∞
i = j = 1
while (k ≤ p) and (L[i] ≠ R[j])
      if L[i] < R[j]
            A[k] = L[i]
            i = i + 1
      else // L[i] > R[j]
            A[k] = R[j]
            j = j + 1
      k = k + 1
return k ≤ 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[i..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)
n = A.length
i = j = 1
while (i ≤ n) and (A[i] + A[j] ≠ key)
    if j = n
        i = i + 1
    else
        j = j + 1
return i ≤ 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)
n = A.length
sort(A) // complessità O(n log n)
i = 1, j = n
while (i ≤ j) and (A[i] + A[j] ≠ key)
    if A[i] + A[j] < key
        i = i + 1
    else
        j = j - 1
return i ≤ 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 $A[i] + A[j] < key$

$\rightarrow$ incremento i;
(1) **non** cambia;
(2) (vogliamo dimostrare) $\forall k \in [i, n] \quad A[i] + A[k] \neq key$.
Distinguiamo 2 casi.

- 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

$$A[i] \geq 0 \quad \forall i \in [i, n],\ key \leq top$$
$$V[v] = \texttt{true} \text{ sse } \exists i : A[i] = v$$

$\textsc{SumV}(A, key)$

$V[0\,..\,key] \leftarrow \textsc{false}$ // $\Theta(key) = O(top) = O(1)$
$i = 1$
$found = \textsc{false}$
while $(i \leq n)$ **and** *not found*
    if $A[i] \leq key$
        $V[A[i]] = \textsc{true}$
        $found = V[key - A[i]]$
    $i = i + 1$
return *found*

Complessità:

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

A.6 Heapsort

Per approfondire, vedi 8.1.

```
Left(i)
    // restituisce il figlio sx del nodo i
    return 2 * i
```

```
Right(i)
    // restituisce il figlio dx del nodo i
    return 2 * i + 1
```

```
Parent(i)
    // restituisce il genitore del nodo i
    return ⌊i/2⌋
```

```
MaxHeapify(A, i)
    l = Left(i)
    r = Right(i)
    if (l ≤ A.heapsize) and (A[l] > A[i])
    else
        max = i
    if (max ≠ i)
        A[i] ↔ A[max]
        MaxHeapify(A, max)
```

```
BuildMaxHeap(A)
    A.heapsize = A.length
    for i = ⌊A.length/2⌋ down to 1
        MaxHeapify(A, i)
```

```
HeapSort(A)
    BuildMaxHeap(A) // O(n)
    for i = A.length down to 2
        A[1] ↔ A[i]
        A.heapsize = A.heapsize − 1
        MaxHeapify(A, 1) // O(log n)
```

A.7 Code con priorità

(Sezione 8.2)

```
Max(A)
if A.heapsize = 0
    error
else return A[1]
```

```
ExtractMax(A)
max = A[1]
A[1] = A[A.heapsize]
A.heapsize = A.heapsize − 1
MaxHeapify(A, 1) // ripristina le proprietà di MaxHeap
return max
```

```
MaxHeapifyUp(A, i)
if (i > 1) and (A[i] > A[Parent(i)])
    A[i] ↔ A[Parent(i)]
    MaxHeapifyUp(A, Parent(i))
```

```
Insert(A, x)
A.heapsize = A.heapsize + 1
A[A.heapsize] = x
MaxHeapifyUp(A, A.heapsize)
```

```
IncreaseKey(A, i, δ)
// Precondizione: δ ≥ 0
A[i] = A[i] + δ
MaxHeapifyUp(A, i)
```

```
ChangeKey(A, i, δ)
A[i] = A[i] + δ
if δ > 0
    MaxHeapifyUp(A, i)
else // δ ≤ 0
    MaxHeapify(A, i)
```

```
DeleteKey(A, i)
old = A[i]
A[i] = A[A.heapsize]
A.heapsize = A.heapsize − 1
if old ≤ A[i]
    MaxHeapifyUp(A, i)
else
    MaxHeapify(A, i)
```

B Esercizi

B.1 Ricorrenze

- $T(n) = aT(n-1) + b \qquad a, b > 1$

  · *radice*: costo b;

  · la radice ha a figli di costo b;

  · ...

  · foglie terminali $O(1)$.

  Esplicitando il caso base della ricorrenza otteniamo:

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

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

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

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

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

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