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 maggio 2019

Indice

1 Introduzione

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 → **Algoritmo** → 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.

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

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 Analisi

- Tempo di esecuzione;
- Spazio (memoria);

- Correttezza;
- Manutenibilità.

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.

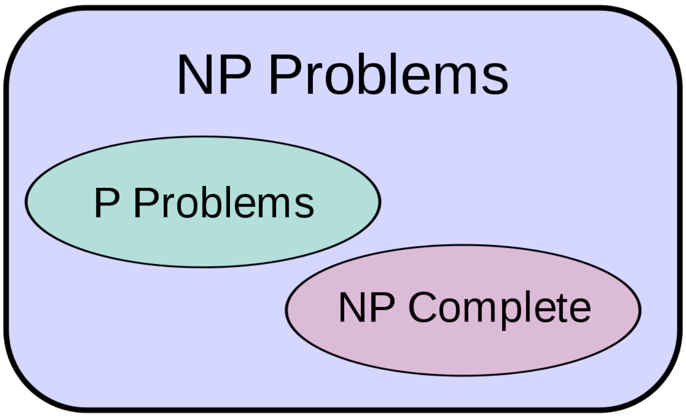


Figura 1: Complessità T(n).

2 Algoritmi di Ordinamento

2.1 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`.

2.2 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: | 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`: $A[1,..,n]$, $A.length$.

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

Pseudocodice Segue lo pseudocodice dell'`Insertion Sort`.

```
INSERTIONSORT(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]` $\leq$ `key)`: `A[i+1] = key`.

2.2.1 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.

2.2.2 Complessità di Insertion Sort

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

```
InsertionSort(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:

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

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

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

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

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

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

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

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

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

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

$\boldsymbol{t_j}$ 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 (2.2.2)

b) Caso peggiore (2.2.2)

c) Caso medio (2.2.2)

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.

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 $\displaystyle\sum_{j=2}^{n} j$ e di $\displaystyle\sum_{j=2}^{n}(j-1)$, 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 $T^{IS}_{max}(n)$

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

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`.

```
MergeSort(A, p, r)
if p < r
    q = ⌊(p+r)/2⌋ // arrotondato per difetto
    MergeSort(A, p, q) // ordina A[p..q]
    MergeSort(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
```

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]`.

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

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

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

Per ipotesi induttiva:

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

2.4.1 Approfondimento sull'Induzione

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.

Induzione completa

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

2.4.2 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:

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

$$\Downarrow$$

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

con

[1] Il simbolo # verrà usato per indicare la cardinalità di un insieme.

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

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

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

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

$$T^{MS}(n_1) \qquad\qquad T^{MS}(n_2)$$
$$an_1 + b \qquad\qquad an_2 + b$$

$$T^{MS}(n_{11}) \qquad T^{MS}(n_{12}) \qquad T^{MS}(n_{21}) \qquad T^{MS}(n_{22})$$
$$an_{11} + b \qquad an_{12} + b \qquad an_{21} + b \qquad an_{22} + b$$

$$\dots$$

$$c_0 \qquad c_0 \qquad \dots \qquad \dots \qquad \dots \qquad c_0 \qquad c_0$$

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.

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

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

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

...

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

Poniamo $n = 2^h$. Abbiamo

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

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

2.5 Confronto tra Insertion Sort e Merge Sort

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

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	$30s$

3 Limiti Asintotici e Ricorrenze

3.1 Notazione Asintotica

Il **tempo di esecuzione** è difficile da calcolare, come visto nella sezione 2.2.2. 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 $f(n)$.

3.1.1 Limite Asintotico Superiore

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

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

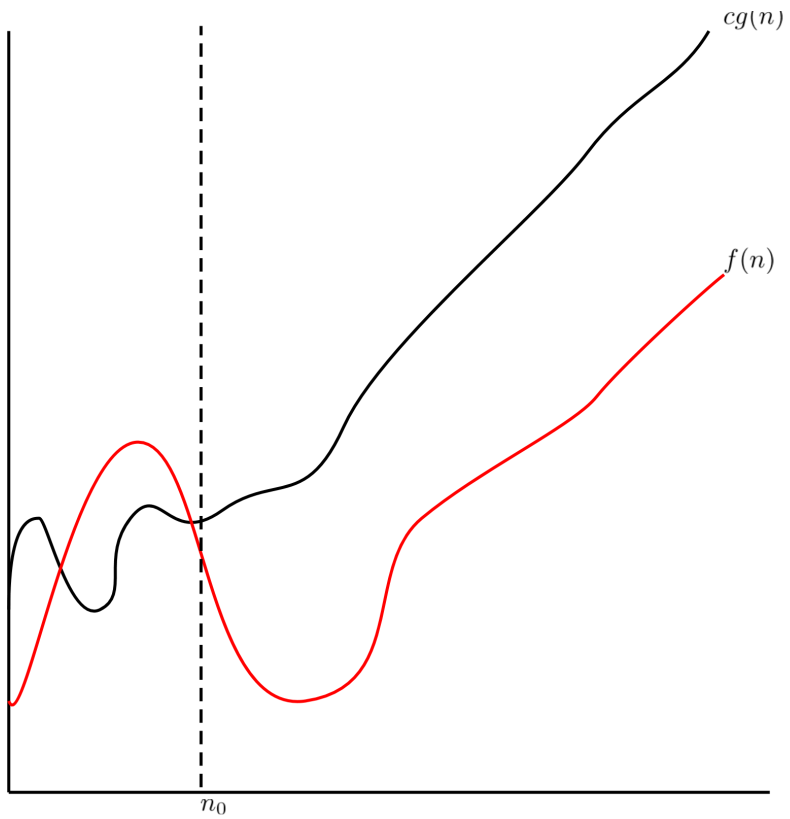


Figura 2: Rappresentazione del limite asintotico superiore per $f(n)$

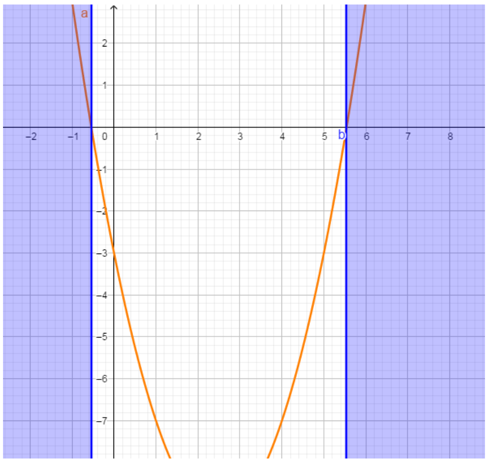
Esempi

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

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

3.1.2 Limite Asintotico Inferiore

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

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

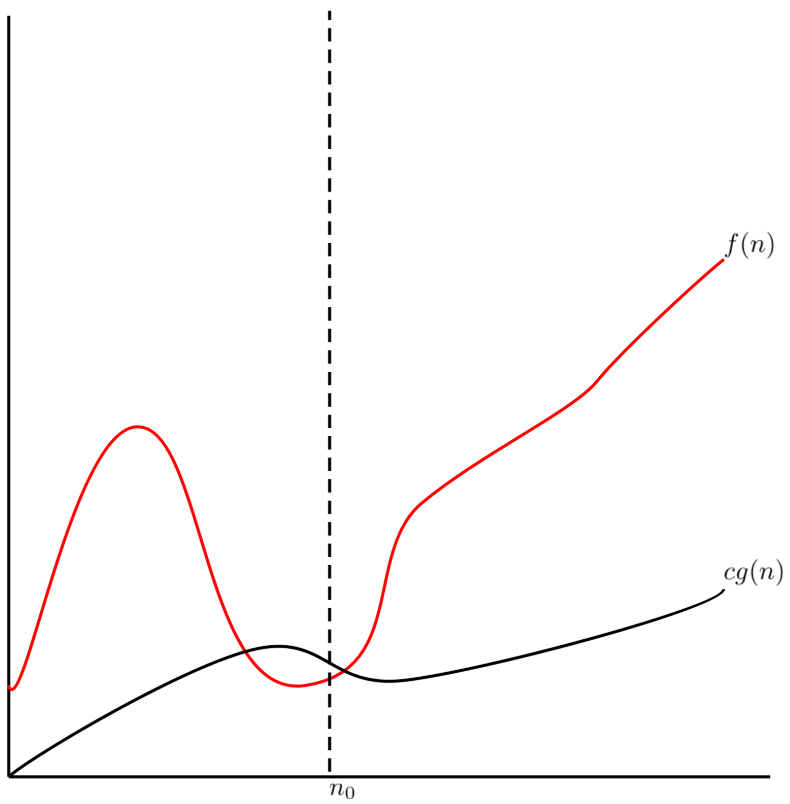


Figura 3: Rappresentazione del limite asintotico inferiore per $f(n)$

Esempi

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

  Deve valere:

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

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

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

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

3.1.3 Limite Asintotico Stretto

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

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

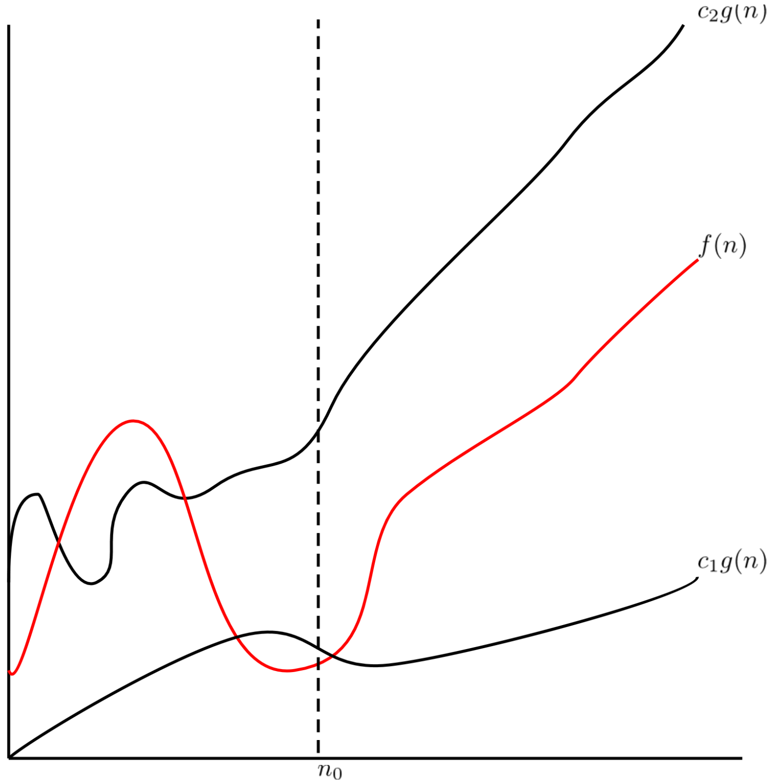


Figura 4: Rappresentazione del limite asintotico stretto per $f(n)$

Esempi

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

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

3.2 Metodo del Limite

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

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

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

Dimostrazione

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

$$\begin{aligned}
&\Rightarrow \forall\ \varepsilon > 0 \quad \exists\ n_0 : \forall\ n \geq n_0 \quad \left|\frac{f(n)}{g(n)} - k\right| \leq \varepsilon \\
&\Rightarrow -\varepsilon \leq \frac{f(n)}{g(n)} - k \leq \varepsilon \\
&\Rightarrow k - \varepsilon \leq \frac{f(n)}{g(n)} \leq k + \varepsilon \\
&\Rightarrow (k-\varepsilon)g(n) \leq f(n) \leq (k+\varepsilon)g(n) \qquad \text{per } 0 < \varepsilon < k \\
&\Rightarrow f(n) = \Theta\big(g(n)\big)
\end{aligned}$$

2. Sia $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$

$$\begin{aligned}
&\Rightarrow \forall\ \varepsilon > 0 \quad \exists\ n_0 : \forall\ n \geq n_0 \quad \frac{f(n)}{g(n)} \leq \varepsilon \\
&\Rightarrow f(n) \leq \varepsilon g(n) \\
&\Rightarrow f(n) = O\big(g(n)\big)
\end{aligned}$$

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

$$\begin{aligned}
&\Rightarrow \exists\ c > 0 \quad \exists\ n_0 \in \mathbb{N} : \forall\ n \geq n_0 \quad cg(n) \leq f(n) \\
&\Rightarrow \text{Impossibile, infatti sia } \varepsilon = \frac{c}{2} > 0 \\
&\Rightarrow \exists\ n_1 \in \mathbb{N} : \forall\ n \geq n_1 \quad f(n) \leq \frac{c}{2}g(n) < cg(n) \\
&\Rightarrow \text{Contraddizione!}
\end{aligned}$$

3. Si dimostra similmente al punto (2)

3.3 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

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

3.4 Complessità di un Problema

Dato un problema P {INPUT $\rightarrow$ OUTPUT}, la **complessità** di P è la complessità dell'algoritmo più efficiente che risolve P.

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

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

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

3.4.1 Esempio: limite inferiore per ordinamento basato su scambi di elementi contigui

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 $k+1$).

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

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.

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

- $i = k$ 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.

$\implies$ qualunque algoritmo di ordinamento basato su scambi è $\Omega\left(\frac{n(n-1)}{2}\right) = \Omega(n^2)$

3.5 Soluzione di Ricorrenze

Abbiamo visto per `Merge Sort` la complessità nel modo seguente:

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p+r)/2⌋
    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** (3.5.1);
- **Master Theorem** (3.5.2).

3.5.1 Metodo di Sostituzione

Dato una ricorrenza, si può provare a "indovinare" la soluzione e dimostrare che è corretta, 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.

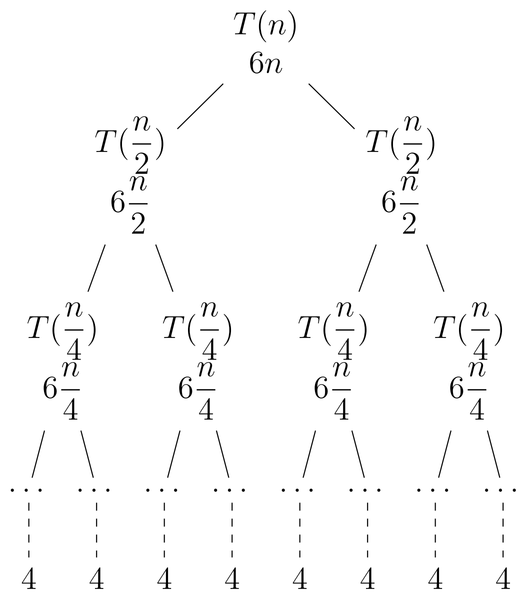
Esempio

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

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

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

Costruendo l'albero delle ricorrenze si intuisce già la soluzione:



Tutte le foglie insieme costano $4n$. Ogni altro livello dell'albero costa $6n$, ma siccome l'altezza è circa $\log_2 n$, in totale costano $6n \log_2 n$.

Per essere sicuri della soluzione, facciamo il procedimento per intero. Proviamo a "indovinare" la soluzione. Assomiglia a `Merge Sort`, quindi ipotizziamo abbia una complessità con un andamento simile

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

Facciamo la prova induttiva.

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

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 $T(n)$

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

$$
\begin{aligned}
b + 6 - a = b &\Rightarrow a = 6 \\
2c = c &\Rightarrow c = 0 \\
b + c = 4 &\Rightarrow b = 4 \\
T(n) &= an \log n + bn + c \\
&= 6n \log n + 4n
\end{aligned}
$$

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).

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

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

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

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

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 ...

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

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 ...

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

3.5.2 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
$$T(n) = \Theta\big(f(n)\big)$$

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

B)

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

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

C)

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

Esercizi

- $T^{MS} = 2T\left(\frac{n}{2}\right) + a'n + b'$
  Abbiamo (rispetto alla forma $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$)

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

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

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

- $T(n) = 5T\left(\frac{n}{2}\right) + 2n^2 + n\log n$
  Abbiamo (rispetto alla forma $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$)

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

- $T(n) = 5T(\frac{n}{2}) + n^3$ 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**?

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

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

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

  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 $O(1)$.

$$
\begin{aligned}
T(n) &= \sum_{j=0}^{\log_3 n} n^3 \log \frac{n}{3^j} = n^3 \sum_{j=0}^{\log_3 n} (\log n - j \log 3) + cn = \\
&= n^3(\log n)^2 - n^3 \log 3 \sum_{j=0}^{\log_3 n} j + cn \qquad \Big(\sum_{j=0}^{\log_3 n} j \cong (\log_3 n)^2 \Big) \\
T(n) &= 27T\left(\frac{n}{3}\right) + n^3 \log n \\
T(n) &= \Theta\big(n^3(\log n)^2\big) \qquad \text{ipotesi ricavata}
\end{aligned}
$$

Devo dimostrare che valgano le seguenti condizioni:

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

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

$$
\begin{aligned}
T(n) &\le c \cdot n^3\big(n^3(\log n)^2\big) \qquad c > 0 \\
T(n) &= 27T\left(\frac{n}{3}\right) + n^3 \log n \\
&\left(\text{ipotesi induttiva } T\left(\frac{n}{3}\right) \le c \cdot \left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 \right) \\
&\le 27c\left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 + n^3 \log n = \\
&= \frac{\cancel{27}cn^3}{\cancel{27}}(\log n - \log 3)^2 + n^3 \log n = \\
&= cn^3\Big((\log n)^2 - 2\log 3 \log n + (\log 3)^2\Big) + n^3 \log n = \\
&= cn^3(\log n)^2 - n^3\Big(\log n(2c\log 3 - 1) - c(\log 3)^2\Big) \\
&\le cn^3(\log n)^2
\end{aligned}
$$

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

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

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

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

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

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

4 Algoritmi di Ordinamento (cont.)

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`:

$$\textit{input} + 1 \text{ variabile} \Rightarrow \text{spazio } \textbf{costante } \Theta(1) \text{ (detto ``in loco'')}$$

- `Merge Sort`: spazio con costo lineare.

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

4.1 Heap Sort

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 e le foglie presenti sono addossate a sinistra.

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⌋
```

4.1.1 Max Heap

Max Heap è uno heap che soddisfa la seguente proprietà:

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

Equivalentemente

$$\begin{gathered}\forall \text{ nodo } A[i],\\ A[i] \leq \text{antenati}\\ \Downarrow\\ A[i] \leq A[\mathbf{Parent(i)}]\end{gathered}$$

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

Correttezza di MaxHeapify

- **Casi base**: $max = i$, distunguo due casi:
  - i è foglia ($l, r > A.heapsize$);
  - $A[i] \geq A[l], A[r]$;
- **Induzione**: $max \neq i$, distinguo due casi:
  - $max = l,\ A[l] \geq A[i], A[r]$;
  - $max = r,\ A[r] \geq A[i], A[l]$.

Complessità $O(h)$, con h altezza del sotto-albero radicato in i

$$\begin{aligned} & n \geq (2^{(h-1)+1}) + 1 = 2^h \\ & h \leq \log_2 n \\ & \quad \Rightarrow O(h) \cong O(\log n) \end{aligned}$$

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

Quali sono i nodi foglia?

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

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

- 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)
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 complessità $O(\log n)$), ottenendo una complessità finale $O(n \log n)$, 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.
⇒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($A, 1$) // $O(\log n)$

Complessità $O(n \log n)$.

4.1.2 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 valere 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[Parent(i)])
      A[i] ↔ A[Parent(i)]
      MaxHeapifyUp(A, 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**.

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

Ecco ora lo pseudocodice della funzione `Insert`.

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

`Insert` ha complessità $O(\log n)$, la stesso di `MaxHeapifyUp`.

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

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

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

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

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

```
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 complessità $O(\log n)$.

4.2 Quick Sort

Il Quicksort è probabilmente l'algoritmo di ordinamento più utilizzato e nella pratica efficiente, nonostante abbia un caso pessimo di $O(n^2)$.

- Caso pessimo $O(n^2)$;
- Caso medio e migliore $O(n \log n)$;
- costanti basse.

Si basa sul paradigma del *divide et impera*:

- *Divide*
  - → Secglie un *pivot* x in `A[p, r]`;
  - → partiziona in `A[p, q-1]` $\leq x$ e `A[q+1, r]` $\geq x$;
- *Impera*

  Ricorre su `A[p, q-1]` e `A[q+1, r]`;
- *Combina*

  (Non fa nulla).

Pseudocodice Segue lo pseudocodice del `Quicksort`.

```
QuickSort(A, p, r)
if p < r
    q = Partition(A, p, r)
    QuickSort(A, p, q)
    QuickSort(A, q + 1, r)
```

```
Partition(A, p, r)
x = A[r] // pivot A[r]
i = p − 1
// A[p, i] ≤ x
// A[i+1, j-1] > x
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        A[i] ↔ A[j]
A[i + 1] ↔ A[r]
return i + 1
```

4.2.1 Correttezza di QuickSort(A, p, r)

Caso base array già ordinato, 0 o 1 elemento.

Induzione Abbiamo, dopo `Partition`

$\leq$ `A[q]`	`A[q]`	$\geq$ `A[q]`

`QuickSort(A, p, q)`	$\leq$ `A[q]`, ord	`A[q]`	$>$ `A[q]`
`QuickSort(A, q+1, r)`	$\leq$ `A[q]`, ord	`A[q]`	$>$ `A[q]`, ord

Esempio Dato l'array A, scelgo come **pivot** x l'ultimo elemento.

9	6	0	8	4	2

pivot: $\boxed{2}$

`i` punta alla cella 0 (ossia nessuna cella)
`j` punta alla cella 1: $\boxed{9}$
$9 > 2$? Sì $\Rightarrow$ `j++`
$6 > 2$? Sì $\Rightarrow$ `j++`
$0 > 2$? No $\Rightarrow$ `i++`, `A[i]` $\leftrightarrow$ `A[j]`, `j++`

0	6	9	8	4	2

pivot: $\boxed{2}$

`i` punta alla cella 1: $\boxed{0}$
`j` punta alla cella 4: $\boxed{8}$
$8 > 2$? Sì $\Rightarrow$ `j++`
$4 > 2$? Sì $\Rightarrow$ `j++`
Scambio `A[i+1]` con `x`, ottenendo

0	2	9	8	4	6

I primi due $(i + 1)$ elementi sono ordinati:

0	2

Chiamo ricorsivamente `QuickSort` con $q = i + 1$.

9	8	4	6

pivot: $\boxed{6}$

`i` punta alla cella 0 (ossia nessuna cella)
`j` punta alla cella 1: $\boxed{9}$
$9 > 6$? Sì $\Rightarrow$ `j++`
$8 > 6$? Sì $\Rightarrow$ `j++`
$4 > 6$? No $\Rightarrow$ `i++`, `A[i]` $\leftrightarrow$ `A[j]`, `j++`

4	8	9	6

pivot: 2

`i` punta alla cella 1: 4
`j` punta alla cella 4: 6, quindi ho finito.
Scambio `A[i+1]` con `x`, ottenendo

4	6	9	8

I primi due $(i+1)$ elementi sono ordinati:

4	6

Chiamo ricorsivamente `QuickSort` con $q = i + 1$.

9	8

pivot: 8

`i` punta alla cella 0 (ossia nessuna cella)
`j` punta alla cella 1: 9
$9 > 8$? Sì $\Rightarrow$ `j++`
Ho finito, scambio `A[i+1]` con `x`, ottenendo

8	9

Guardando l'array completo ottengo il risultato atteso:

0	2	4	6	8	9

4.2.2 Complessità di QuickSort

`Partition` costa $\Theta(n)$

$$T^{QS} = \Theta(n) + T^{QS}(q-p) + T^{QS}(n-(q-p)-1)$$
$$q - p < n$$

Caso peggiore

$$\begin{aligned} T^{QS} =& \Theta(n) + T^{QS}(n-1) = \Theta(n^2) \qquad (\Theta(n) = cn) \\ & T(n) \\ & cn \\ & cn-1 \\ & cn-2 \\ & \dots \\ & d \end{aligned}$$

$$\sum_{j=1}^{n-1} c(n-j) + d = \sum_{k=1}^{n} ck + d = \\ = c\sum_{k=1}^{n} k + d \qquad \left(\frac{c(n+1)n}{2} + d = \Theta(n^2)\right)$$

$$T(n) = \Theta(n^2) \Rightarrow \begin{cases} = O(n^2) \\ = \Omega(n^2) \end{cases}$$

- $T(n) = O(n^2)$

$$\begin{aligned} T(n) &= O(n^2) \Rightarrow T(n) \leq cn^2 \qquad \forall\ n \geq n_0,\ c > 0 \\ &= T(n-1) + \Theta(n) \leq dn \\ &\leq c(n-1) + dn \\ &= cn^2 - 2cn + c + dn \leq cn^2 \\ &\quad 2cn - dn - c \geq 0 \\ &\quad n(2c-d) - c \geq 0 \qquad \text{ok, } c > \frac{d}{2} \\ T(n) &= O(n^2) \end{aligned}$$

- $T(n) = \Omega(n^2)$ analogo.

Caso migliore

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

Caso medio Qualunque partizionamento proporzionale da complessità $\Theta(n \log n)$, come ad esempio

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) = \Theta(n \log n)$$

Solo il caso in cui una delle due partizioni è costante, si ricade nel caso pessimo. Per ovviare al problema, si può utilizzare una versione di `Partition` che rende impossibile il partizionamento costante.

RandomizedPartition(A, p, r)
q = Random(p, r)
$A[q] \leftrightarrow A[r]$
return Partition(A, p, r)

4.3 Quick Sort a Tre Partizioni

`Quicksort` con `RandomizedPartition` funziona bene ed evita, quasi in ogni circostanza, di imbattersi nel caso pessimo, ad eccezione di un caso particolare: se in **input** viene dato un array con tutti gli elementi uguali, si ottiene il temuto caso pessimo $O(n^2)$.

Per ovviare al problema, è sufficiente partizionare Quicksort in tre partizioni invece di due. Dato un **pivot** x, partizioniamo A nel seguente modo:

$< x$	$= x$	$> x$

Durante l'algoritmo, la disposizione sarà questa:

$< x$	$= x$		$> x$

(La cella vuota è la regione ancora da esplorare).

```
TRIPARTITION(A, p, r)
x = A[r]
i = p − 1
k = p
j = r
while k < j
    if A[k] < x
        i = i + 1
        A[i] ↔ A[k]
        k = k + 1
    else if A[k] > x
        j = j − 1
        A[j] ⇔ A[k]
    else
        k = k + 1
// k = j
A[j] ⇔ A[r]
return (i + 1, j) // restituisce una coppia di valori
```

```
QUICKSORT(A, p, r)
if p < r
    q1, q2 = TRIPARTITION(A, p, r)
    QUICKSORT(A, p, q1 − 1)
    QUICKSORT(A, q2 + 1, r)
```

4.4 Limite Inferiore

`Input`: $a_1 \dots a_n$

`Output`: permutazione $a'_1 \dots a'_n$ tale che

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

Confronti e assegnamenti Osservazioni:

→ Se "conto" solo alcune operazioni il limite inferiore vale in generale. Consideriamo solo l'operatore di confronto;

→ Elementi tutti distinti ($a_i \neq a_j$ se $i \neq j$), l'operatore di confronto == restituisce sempre FALSE.

4.5 Albero di Decisione

È una rappresentazione "astratta" delle possibili esecuzioni di un algoritmo di ordinamento su un input di dimensione fissata $A[1 \dots n]$.

→ nodi interni:

$$i \ : \ j \Rightarrow \text{confronta } A[i] \leq A[j]$$

→ **foglie** (ogni foglia è una possibile permutazione)

Rivediamo una versione di `Insertion Sort` basato su scambi.

INSERTION-SORT(A)

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

Ecco un esempio di **Albero di Decisione** per l'array $A[a_1, a_2, a_3]$ con

$$a_1 = 1,\ a_2 = 2,\ a_3 = 3$$

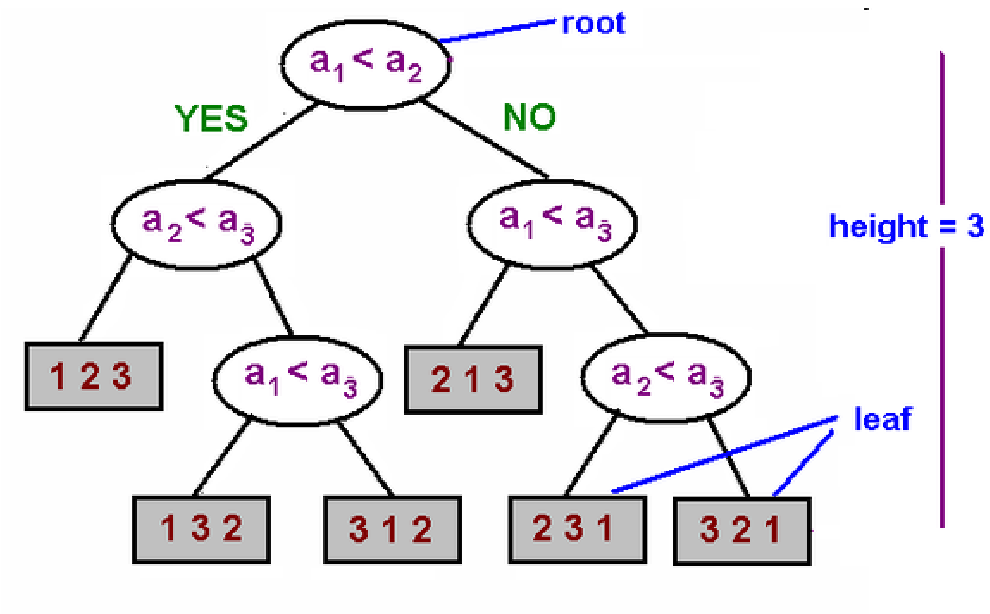


Figura 5: Albero di decisione per l'array $A[1, 2, 3]$

Osservazione

Altezza dell'albero di decisione = limite inferiore per caso pessimo

$$\begin{array}{ll} \text{per IS} & n^2 \\ \text{per MS} & n \log n \end{array}$$

In generale, le foglie contengono tutte le permutazioni.

$$\#foglie \geq n! \qquad (\#foglie \leq 2^h)$$

$$\begin{aligned} h &\geq \log_2 n! \\ &\geq \log_2 \left(n(n-1)(n-2)\dots\frac{n}{2} \right) \\ &\geq \log_2 \left(\frac{n}{2} \left(\frac{n}{2} - 1 \right) \left(\frac{n}{2} - 2 \right) \dots \frac{n}{2} \right) = \\ &= \log_2 \left(\frac{n}{2} \right)^{\left(\frac{n}{2}\right)} = \frac{n}{2} \left(\log_2 n - \log_2 2 \right) = \frac{n}{2} (\log_2 n - 1) = \Theta(n \log n) \end{aligned}$$

4.6 Counting Sort

Esistono degli algoritmi di ordinamento che, in certe condizioni e per certi input, permettono di ordinare in tempo lineare $\Omega(n)$

Assumo

- interi;
- in $[0, k]$

`Input`: $A[1 \mathinner{..} n]$ con $A[j] \in [0, k] \forall\ j$;

`Output`: $B[1 \mathinner{..} n]$ permutazione ordinata di A;

`Supporto`: $C[0 \mathinner{..} k]$.

Counting-Sort(A, B, k)

$C[0 \mathinner{..} k] \leftarrow 0$
for $j = 1$ **to** $A.\mathit{length}$
  // $C[x] = \#\mathit{elem}$ in A con valore x
  $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
  // $C[x] = \#\mathit{elem}$ in A con valore $\leq x$
  $C[i] = C[i-1] + C[i]$
for $j = A.\mathit{length}$ **to** 1
  $B[C[A[j]]] = A[j]$
  $C[A[j]] = C[A[j]] - 1$

Costo?

$C[0, k] \leftarrow 0$	$\Theta(k)$
`for j=1...`	$\Theta(n)$
`for i=1...`	$\Theta(k)$
`for j=A.length...`	$\Theta(n)$

Somma $\Theta(n + k)$ con $k = \Theta(1) \Rightarrow \Theta(n)$

Problema di memoria Il problema di `Counting Sort` è la memoria. Infatti, al crescere di k, la memoria richiesta per allocare `C` cresce esponenzialmente.

Dimensione k	Memoria occupata da `C[]`
1 Byte = 8 bit	2^8Bytes = 256Bytes
2 Bytes = 16 bit	2^{16}Byte $\cdot$ 2Bytes = 256Megabytes
8 Bytes = 64 bit	2^{64}Byte $\cdot$ 8Bytes = 512Terabytes

4.7 Radix Sort

Il Radix Sort è un algoritmo di ordinamento in tempo lineare $O(n)$, come `Counting Sort`, che risolve i problemi di memoria di quest'ultimo.

L'idea è quella di ordinare cifra per cifra, dalla cifra meno significativa alla più significativa con un algoritmo **stabile**[1].

(iniziale)	(terza cifra)	(seconda cifra)	(prima cifra)
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

`Input`: `A[1..n]` con `A[i]` di d cifre e base b, `A[i]`$= a_d a_{d-1} \dots a_1$.

RADIXSORT(A, d)

```
1  for j = 1 to d
2      ordina A rispetto alla cifra j // A^j[i] = a_j a_{j-1} ... a_1
       con COUNTINGSORT // A^{j-1} ordinato
```

Correttezza di RadixSort(A, d)

- **Inizializzazione**: ok;
- **Mantenimento**: se A^{j-1} è ordinato e ordino rispetto alla j-esima cifra con un algoritmo stabile, allora A^j è ordinato.

$$i < i' \Rightarrow A^j[i] \leq A^j[i']$$

$$\text{Siano } A^j[i] = a_j a_{j-1} \dots a_1$$
$$A^j[i'] = a'_j a'_{j-1} \dots a_1$$

Posso distinguere due casi:

[1] Un algoritmo di ordinamento stabile è un algoritmo che ordina e non scambia mai due chiavi se non è necessario (e.g. due celle con la stessa chiave).

1.

$$\begin{aligned} a_j \neq a'_j &\Rightarrow a_j < a'_j \\ &\Rightarrow A^j[i] < A^j[i'] \end{aligned}$$

2.

$$\begin{aligned} a_j = a'_j &\Rightarrow A^j[i] \leq A^j[i'] \qquad \text{(stabilità)} \\ &\Rightarrow A^j[i] = a_j A^j[i] \leq \\ &\leq A^j[i'] = a'_j A^{j-1}[i'] \end{aligned}$$

Costo?

$$d \text{ volte } \texttt{CountingSort} \;\; \Theta(n+b) \Rightarrow \Theta\big(d(n+b)\big) = \Theta(n)$$
$$\text{con } d = \Theta(1), \;\text{ base } b = \Theta(n)$$

5 Tabelle Hash

$$U \text{ universo delle chiavi}$$
$$U = \{0, 1, \ldots, |U| - 1\}$$
$$T[0\,..\,|U|-1] \text{ tabella hash}$$

$$T[k] \text{ contiene} \quad \begin{cases} \text{elemento } x \text{ con } x.key = k & \text{se c'è} \\ \perp & \text{altrimenti} \end{cases}$$

INSERT(T, x)
1 $T[x.key] = x$ // $\Theta(1)$

DELETE(T, x)
1 $T[x.key] = nil$ // $\Theta(1)$

SEARCH(k)
1 **return** $T[k]$ // $\Theta(1)$

Problema e.g. consideriamo che la **key** sia di 8 caratteri (e 8 bit per rappresentare un carattere). Risulta molto costosa in termini di memoria la tabella hash.

$$2^8 \ldots 2^8$$
$$(2^8)^8 = 2^{64} \cong 10^{19}$$

Idea

$$U = \{0, 1, \ldots, |U| - 1\}$$
$$T[0 \ldots m-1] \qquad m << |U|$$

La "traduzione" per ottenere $x.key$ da x cosa comporta?

$$h : U \to \{0, 1, \ldots, m-1\} \text{ funzione di } \textbf{hashing}$$
$$n = \#elem \text{ memorizzati nella tabella } T$$

Se $n > m$, esisteranno $x_1, x_2 : h(x_1.key) = h(x_2.key)$.

Abbiamo **due soluzioni**:

1. **Chaining** (5.1);
2. **Open Addressing** (5.2).

5.1 Chaining

Il **Chaining** propone come soluzione quella di mettere sulla tabella liste dinamiche di elementi, invece che singoli elementi, in modo che in caso si incorra in una cella già occupata dopo un **hashing**, l'elemento venga inserito in coda (o in testa) alla lista.

Idea `T[i]` = lista elementi x tali che $h(x.key) = i$

Insert(T, x)
1 Inserisci x nella lista $T[h(x.key)]$ // $O(1)$

Delete(T, x)
1 Delete x from $T[h(x.key)]$ // $O(1)$

Search(T, k)
1 Cerco in $T[h(k)]$ un elemento
x con chiave k // $O(n)$

`Search` ha una complessità di $O(n)$, e questo è inaccettabile.

$$\begin{gathered} n = \#\text{elementi inseriti} \\ m = \text{dimensione di } T \\ \alpha = \frac{n}{m} \quad \text{fattore di carico} \\ \alpha \text{ può essere } <, = \text{ oppure } > \text{ di } 1 \end{gathered}$$

5.1.1 Hashing uniforme semplice

Ogni elemento di **input** è "mandato" da h con la stessa probabilità $\left(\frac{1}{m}\right)$ in una delle m celle.

Caso medio $\Theta(1 + \alpha)$, 1 è l'accesso alla tabella.

Consideriamo $n_1, n_2, \ldots, n_{m-1}$ la lunghezza delle m liste. La lunghezza attesa di una lista è:

$$E[n_j] = \sum_{i=1}^{n} \frac{1}{m} \cdot 1 = \frac{n}{m} = \alpha$$

Ricerca di una chiave La chiave può essere:

- **Assente**. `Search(k)`, `k` non c'è.
  - Calcolo `h(k)` $\rightarrow (\Theta(1))$;
  - Accedo a `T[h(k)] = j` $\rightarrow (\Theta(1))$;
  - Scorro n_j elementi $(n_j = \alpha) \rightarrow (\Theta(\alpha))$.

  Nel complesso, ho $\Theta(1+\alpha)$
- **Presente**. `Search(k)`, `k` presente.
  - `h(k)` e `T[h(k)]`

Se $x_1, x_2, \ldots x_n$ sono gli elementi inseriti

Costo della ricerca di x_i:

$$\begin{aligned}
&1 + \#elem \quad x_j : j > 1,\ h(x_i.key) = h(x_j.key) \\
&= 1 + \sum_{j=i+1}^{n} (prob\ h(x_i.key) = h(x_j.key)) \\
&= 1 + \sum_{j=i+1}^{n} \frac{1}{m} = 1 + \frac{n-i}{m}
\end{aligned}$$

$$\begin{aligned}
&\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right) \\
&= \frac{1}{n}\left(n + \sum_{i=1}^{n}\frac{n-i}{m}\right) = \frac{1}{n}\left(n + \frac{1}{m}\sum_{z=0}^{n-1} z\right) \\
&= 1 + \frac{1}{m\cdot n}\cdot\frac{n(n-1)}{2} = 1 + \frac{n}{2m} - \frac{1}{2m}\cdot\left(\frac{n}{n}\right) \\
&\qquad\qquad\qquad\qquad\qquad \left(\frac{n}{m} = \alpha\right) \\
&= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1+\alpha)
\end{aligned}$$

Se $n \leq c \cdot m$ per qualche costante positiva c allora

$$\alpha \leq c \Rightarrow \Theta(1)$$

- $h : U \implies \{0, 1, \ldots, m-1\} \Rightarrow h(x) = 0$

5.1.2 Funzioni Hash

Una **funzione hash** deve soddisfare la proprietà di **hashing uniforme**, ossia

> "Ogni chiave ha la stesso probabilità $\frac{1}{m}$ di essere mandata in una qualsiasi delle m celle, indipendentemente dalle chiavi inserite precedentemente."

Consideriamo:

- $x \in [0, 1)$ $(0 \leq x < 1)$, x chiave, estratta in modo indipendente dalla distribuzione uniforme (non realistica).
- Allora $h(x) = \lfloor mx \rfloor$ soddisfa la proprietà di **hashing uniforme**.

L'ipotesi di hash uniforme semplice dipende dalle probabilità con cui vengono estratti gli elementi da inserire; probabilità che in genere non sono note. Le funzioni hash che descriveremo assumono che le chiavi siano degli interi non negativi.

Metodo della divisione

$$U = \{0, 1, \ldots, |\cup| - 1\}$$
$$h(k) = k \mod n$$

- $m = 2^p$ caso pessimo;
- $m = 2^p - 1$ caso non buono. 2^p cifre base.

La soluzione migliore è quella di scegliere chiavi lontane dalle potenze di 2, meglio ancora se numeri primi.

Metodo della moltiplicazione

$$k \in U$$
$$0 < A < 1 \text{ fissato}$$
$$h(k) = m(kA \mod 1) \qquad \text{Miglior } A : \frac{\sqrt{5} - 1}{2}$$

$$m = 2^p \quad w = \#\text{bit parola}$$
$$A = \frac{q}{2^w} \quad 0 < q < 2$$
$$\begin{aligned} & m(kA \mod 1) \\ & = m\left(k\frac{q}{2^w} \mod 1\right) \qquad & (\textbf{shift} \text{ di } w \text{ bit, prendo la parte decimale} \\ & & ka \mod 1 \text{ e la moltiplico per } m = 2^p) \end{aligned}$$

5.1.3 Hashing Universale

Per avere una distribuzione più uniforme delle chiavi nelle liste e non dipendente dall'input, possiamo usare la **randomizzazione**.

Insieme H di funzioni di hash. Scelgo randomicamente da H.

Sotto certe ipotesi ottengo per `Search`:

$$\Theta(1+\alpha)$$

Def (Hashing universale) $\forall\ k_1, k_2 \in U\ k_1 \neq k_2$

$$|\{h \in H : h(k_1) = h(k_2)\}| \leq \frac{|H|}{m}$$
$$\frac{|\{h \in H : h(k_1) = h(k_2)\}|}{|H|} \leq \frac{1}{m}$$

Con il **chaining**, H è universale per ogni $k \in U,\ j = h(k)$

$$\text{Costo medio } \Theta(1+\alpha) \begin{cases} k \text{ non è in } T \rightarrow E[n_j] \leq \alpha \\ k \text{ è in } T \rightarrow E[n_j] \leq 1+\alpha \end{cases}$$

5.2 Open Addressing

$h(k, i)$: k è la chiave, i è il tentativo.

Provo con $h(k, 0)$: se capito in una cella occupata, provo con $h(k, 1)$, poi $h(k, 2)$ e così via, fino a che non trovo una cella libera.

Per esplorare tutta la tabella:

$$h(k, 0), h(k, 1), \ldots, h(k, m-1)$$

che è una permutazione di

$$0, 1, \ldots, m-1$$

```
INSERT(T, x)
i = 0
repeat
    j = h(x.key, i)
    if (T[j] = NIL) or (T[j] = DELETED) // posizione libera
        T[j] = x
        return j
    i = i + 1
until i = m
error
```

```
SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j].key = k
        return j
    i = i + 1
until (i = m)  or  (T[i] = NIL)
return NOT FOUND
```

```
DELETE(T, j)
T[j] = DELETED
```

L'**Open Addressing** risulta una soluzione inefficiente in caso avvengano molte cancellazioni.

5.2.1 Hashing uniforme

Per ogni elemento di input, tutte ($m!$) le sequenze di ispezione sono equiprobabili.

5.2.2 Funzioni di Hash

1. **Ispezione lineare**. Sia $h'(k)$ funzione di hash "ordinaria". Se ricado in una cella occupata, mi sposto su quella immediatamente successiva.

$$h(k,i) = (h'(k) + i) \mod m$$

   Caratteristiche:

   - è semplice;
   - poche permutazioni (m dipende solo da $h'(k)$);
   - causa addensamenti di celle occupate (**addensamento primario**).

2. **Ispezione quadratica**. Fisso $h'(k)$.

$$h(k,i) = h'(k) + c_1 i + c_2 i^2 \qquad c_2 \neq 0$$

   Inserimento di k

   $j = h'(k)$
   $i = 0$
   while $(i < m)$ **and** $(T[j] \neq \text{NIL/DELETED})$
   $\quad i = i + 1$
   $\quad j = (j+1) \mod n$

   $(i = 0) \qquad j = h'(k)$

   $(i = 1) \qquad j = (h'(k)+1)/modm$

   $\vdots$

   $(i = l)$

$$\begin{aligned} j &= \left(h'(k) + \sum_{i=1}^{l} i\right) \mod m \\ &= \left(h'(k) + \frac{l(l+1)}{2}\right) \mod m \\ &= \left(h'(k) + \frac{1}{2}l + \frac{1}{2}l^2\right) \mod m \\ m &= 2^p \text{ permutazione} \end{aligned}$$

3. **Doppio Hash**. Fisso $h_1(k)$, $h_2(k)$

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

Osservazioni:

- I salti sono di dimensione $h_2(k)$ all'incrementare di i;
- Ci sono m^2 sequenze di ispezione;
- $h_2(k)$ e m primi tra loro $\quad (MCD = 1)$;
- $i, i < m \quad h(k,i) = h(k,i') \Rightarrow i = i' \quad$ (**iniettività**)

$$h(k, \_) : \{0, \_, m-1\} \to \{0, \_, m-1\}$$

$$\textbf{iniettiva} \Rightarrow \textbf{biiettiva}$$

$$\begin{gathered} h(k,i) = h(k,i') \\ (h_1(k) + ih_2(k)) \mod m = (h_1(k) + i'h_2(k)) \mod m \\ ((i-i')h_2(k)) \mod m = (ih_2(k) - i'h_2(k)) \mod m = 0 \\ (i-i') \mod m = 0 \\ i \geq i' \quad i - i' < m \\ \Rightarrow i - i' = 0 \\ \Rightarrow i = i' \end{gathered}$$

Scelgo $m = 2^p$, $h_2(k) = 1 + 2h_2'(k)$, $h_2'(k)$ qualunque.
es. $h_2(k) = 1 + k \mod m' \quad$ con $m' < m$

Costo? Il costo della `Search` con **hashing uniforme** si può riassumere come segue.

$$0 \leq \alpha = \frac{n}{m} \leq 1$$

Ricerca di una chiave non presente

(a) $\frac{1}{1-\alpha} \quad$ se $\alpha < 1$

(b) $m \quad$ se $\alpha = 1$

Probabilità di ispezionare la i-esima cella

cella	**probabilità**
$i = 0$	1
$i = 1$	prob. cella 0 occupata: $\frac{n}{m}$
$i = 2$	prob. cella 1 occupata: $\frac{n}{m} \cdot \frac{n-1}{m-1}$
$\dots$	
i	$\frac{n}{m} \cdot \frac{n-1}{m-1} \dots \frac{n-i+1}{m-i+1} \leq \alpha \cdot \alpha \dots \alpha = \alpha^i$

Valore atteso per #celle ispezionate

$$1 + \alpha + \alpha^2 + \dots + \alpha^{i-1} + \dots + \alpha^{m-1}$$

(a) $\alpha < 1 \Rightarrow \frac{1-\alpha^m}{1-\alpha} \leq \frac{1}{1-\alpha}$

(b) m

Ricerca di una chiave presente

(a) $\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right) \quad \alpha < 1$

(b) $1 + \log m \quad \alpha = 1$

Finora, ho inserito $x_0, x_1, \dots, x_i, \dots, x_n$.

$$\begin{aligned} &\text{costo } \texttt{Search} \text{ chiave } x_i \text{ presente} \\ &= \text{costo } \texttt{Search} \text{ chiave } x_i \text{ assente} \\ &\text{in } x_0, \dots, x_{i-1} \quad \frac{1}{1-\alpha_i}, \ \alpha_i = \frac{i}{m} \end{aligned}$$

Numero medio:

$$\begin{aligned} &\frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1-\alpha_i} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \qquad \left(\frac{m}{n} = \frac{1}{\alpha}\right) \\ &= \frac{1}{\alpha} \sum_{l=m-n+1}^{m} \frac{1}{l} \qquad (m - i \to m - n + 1) \end{aligned}$$

- Se $\alpha < 1$

$$
\begin{aligned}
&\leq \frac{1}{\alpha}\int_{n-m}^{m}\frac{1}{x}\mathrm{d}x\\
&= \frac{1}{\alpha}\left(\log m - \log(m-n)\right) = \frac{1}{\alpha}\left(\log\frac{m}{m-n}\right)\\
&= \frac{1}{\alpha}\log\frac{1}{\frac{m-n}{m}}\\
&= \frac{1}{\alpha}\log\left(\frac{1}{1-\left(\frac{n}{m}\right)}\right) = \frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)
\end{aligned}
$$

- Se $\alpha = 1$

$$
\begin{aligned}
\sum_{l=1}^{m}\frac{1}{l} &= 1 + \sum_{l=2}^{m}\frac{1}{l} \leq \int_{1}^{m}\frac{1}{x}\mathrm{d}x\\
&= 1 + (\log m - \log 1) = 1 + \log m
\end{aligned}
$$

Confrontiamo le complessità dei due casi.

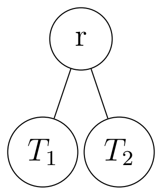
α	$\frac{l}{1-\alpha}$	$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$
$\alpha = 0.3$	1.43	1.19
$\alpha = 0.5$	2.00	1.39
$\alpha = 0.7$	3.33	1.72
$\alpha = 0.9$	10	2.56
$\alpha = 0.99$	100	4.65

6 Alberi di Ricerca

6.1 Alberi Binari di Ricerca (ABR)

Definizione induttiva

- $\varnothing$ è un albero;
- Se r è un nodo, T_1 e T_2 alberi $\Rightarrow r(T_1, T_2)$ è un albero.



Ogni nodo x ha i seguenti campi:

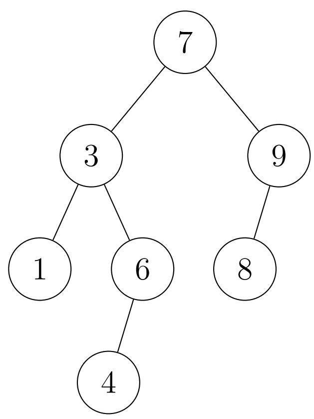
- $x.p$
- $x.key$
- $x.left$
- $x.right$

Proprietà $\forall\, r$

→ Per ogni nodo y in T_1 $y.key \leq x.key$;

→ Per ogni nodo y in T_2 $y.key \geq x.key$.

Esempio Ecco un **albero binario di ricerca** d'esempio:



6.1.1 Visita simmetrica

La visita simmetrica (ordine infisso) visita i nodi in ordine crescente.

In-Order(x)
if $x \neq$ NIL
    In-Order($x.left$)
    Print(x) // $\Theta(1)$
    In-Order($x.right$)

Costo?

$$T(n) = \begin{cases} c & n = 0 \\ T(k) + T(n-k-1) + d & n > 0,\ k < n \end{cases}$$

Stima di complessità: $T(n) = (c+d)n + c$.
Vediamo la dimostrazione (per induzione).

$(n = 0)$ $T(n) = c = (c+d) \cdot 0 + c$

$(n \to n+1)$ $T(n) = T(k) + T(n-k) + d$. Non basta l'induzione ordinaria, usiamo l'**induzione completa**.

$(n > 0)$ Proprietà vera per $n' < n$

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &\quad \text{con } T(k) = (c+d)k + c \text{ e } T(n-k-1) = (c+d)(c-k-1) + c \\ &= (c+d)(\cancel{k} + n - \cancel{k} - 1) + 2c + d \\ &= n(c+d) - c - d + 2c + d \\ &= n(c+d) + c - \cancel{d} + \cancel{d} \\ &\cong \Theta(n) \end{aligned}$$

6.1.2 Ricerca

Ricerca di una chiave k in un albero radicato nel nodo x.

- Se x è NIL $\Rightarrow$ restituisce NIL;
- Altrimenti se $x.key = k \Rightarrow$ restituisce x;
- Altrimenti, ricorre sul prossimo nodo.

```
Search(x, k)
if (x = NIL) or (x.key = k)
    return x
else if k < x.key
    return Search(x.left, k)
else
    return Search(x.right, k)
```

Costo? Nel caso peggiore, il costo è l'altezza dell'albero h ($O(h)$).

Vediamo una versione iterativa di `Search`.

```
Search-it(x, k)
while (x ≠ NIL) or (x.key ≠ k)
    if k < x.key
        x = x.left
    else
        x = x.right
return x
```

Procedura che restituisce il **minimo** di un albero:

```
Min(T)
x = T.root
if x = NIL
    return NIL
else
    while x.left ≠ nil
        x = x.left
return x
```

Procedura che restituisce il **massimo** di un albero.

```
Max(T)
x = T.root
if x = NIL
    return NIL
else
    while x.right ≠ nil
        x = x.right
return x
```

Costo? $O(h)$

6.1.3 Successore di un nodo

Si intende il nodo elencato dopo un nodo x passato come parametro in una visita simmetrica.

Se le chiavi fossero tutte distinte, allora il **successore** di x è il minimo tra i "nodi più grandi di x".

- Se x ha un figlio destro, il **successore** è MIN(x);
- Altrimenti, il successore è l'antenato più vicino di cui x è nel sottoalbero sinistro.

```
SUCCESSOR(x)
if x.right ≠ NIL
    return MIN(x.right)
else
    y = x.p
    while (y ≠ NIL) and (x = y.right)
        x = y
        y = y.p
    return y
```

Costo? $O(h)$

6.1.4 Inserimento

```
INSERT(T, z)
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
z.p = y
if y = NIL
    T.root = z
else
    if z.key < y.key
        y.left = z
    else
        y.right = z
```

Costo? $O(h)$

6.1.5 Eliminazione di un nodo

Distingueremo 2 casi nell'algoritmo:

(1) z ha al più un figlio;

(2) z ha due figli.

Per fare ciò, usiamo una funzione ausiliaria `Transplant`, con costo $O(1)$.

```
TRANSPLANT(T, u, v)
if u.p = NIL
    T.root = v
else
    if u = u.p.left
        u.p.left = v
    else
        u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
Delete(T, z)
if z.left = NIL
    Transplant(T, z, z.right)
else if z.right = NIL
    Transplant(T, z, z.left)
else
    y = Min(z.right)
    if y.p ≠ z
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
        y.left = z.left
        y.left.p = y
        Transplant(T, z, y)
    else
        y.left = z.left
        y.left.p = y
        Transplant(T, z, y)
```

È possibile scrivere `Delete` in modo più compatto.

```
Delete(T, z)
if z.left = NIL
    Transplant(T, z, z.right)
else if z.right = NIL
    Transplant(T, z, z.left)
else
    y = Min(z.right)
    if y.p ≠ z
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
    y.left = z.left
    y.left.p = y
    Transplant(T, z, y)
```

6.2 Red-Black Trees

I **Red-Black Trees** sono ABR i cui nodi hanno un campo **colore** $x.col$, che può essere:

- **R** per il rosso;
- **B** per il nero.

Accorgimento NIL sarà in realtà un nodo, $T.nil$, con $T.nil.col = B$.

Caratteristiche **RB-tree** è in realtà un ABR tale che:

(1) Ogni nodo x ha $x.col \in \{R, B\}$;

(2) La radice $root$ ha $root.col = B$;

(3) Le foglie ($T.nil$) sono B;

(4) Se x è R, i figli sono B;

(5) Per ogni nodo x, ogni cammino da x a una qualsiasi delle foglie ha lo stesso numero di nodi B (calcolato con $bh(x)$).

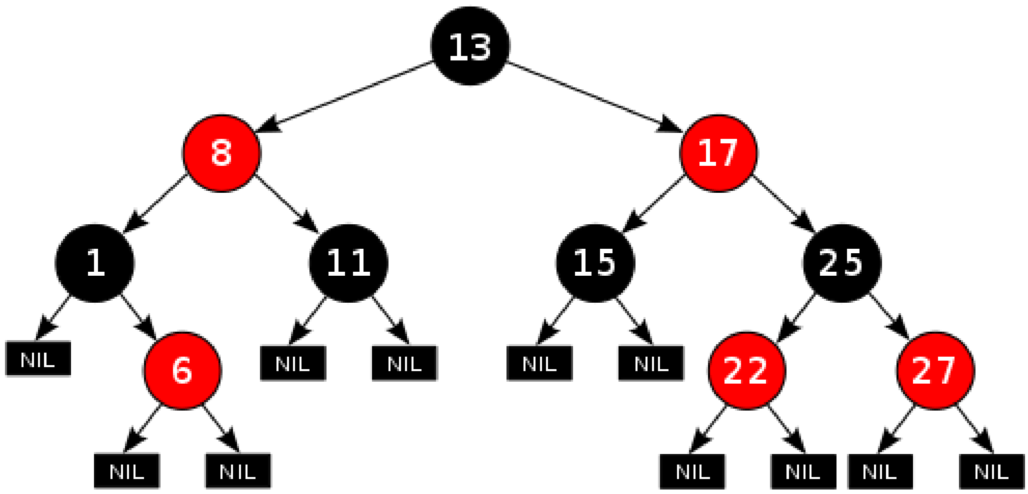


Figura 6: Esempio di un RB-tree.

È possibile notare che:

- In caso non ci fossero nodi rossi, avremo un albero perfettamente bilanciato;
- In ogni cammino, il # di nodi **B** è almeno la metà del # dei nodi **R**

Osservazione Se T è un **RB-tree** con n nodi interni ($\neq$ NIL) e h altezza, allora vale

$$h \leq 2\log(n+1)$$

Dimostrazione Consideriamo

$$n_x \geq 2^{bh(x)} - 1$$

La dimostrazione è per induzione su h_x (altezza del sotto-albero radicato in x).

($h_x = 1$) Allora ho solo $T.nil \Rightarrow n_x = 0 = 2^0 - 1$ $\quad$ (2^0 con $0 = bh(x)$)

($h_x > 1$) Consideriamo x radice. x ha due figli, x_1 e x_2.

Sicuramente vale $h_1, h_2 < h$. Per ipotesi induttiva, valgono:

$$n_{x_1} \geq 2^{bh(x_1)} - 1$$

$$n_{x_2} \geq 2^{bh(x_2)} - 1$$

$$\begin{aligned} n_x &= n_{x_1} + n_{x_2} + 1 \\ &\geq 2^{bh(x_1)} + 2^{bh(x_2)} - 1 \\ &\geq 2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1 \\ &\quad (\text{valgono } bh(x_1) \geq bh(x) - 1,\ bh(x_2) \geq bh(x) - 1) \end{aligned}$$

Complessivamente

$$n = n_{root} \geq 2^{bh(root)} - 1$$

Essendo $bh(root) \geq \frac{h}{2}$, posso ottenere

$$\begin{aligned} n_{root} &\geq 2^{bh(root)} - 1 \\ &2^{\frac{h}{2}} - 1 \\ &\Rightarrow 2^{\frac{h}{2}} \leq n + 1 \\ &\frac{h}{2} \leq \log_2(n+1) \Rightarrow h \leq 2\log_2(n+1) \end{aligned}$$

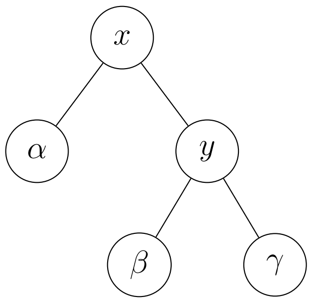
6.2.1 Complessità algoritmi RB-Trees

`Search, Succ, Min, Pred, Max` hanno un costo di $O(h) = O(\log n)$

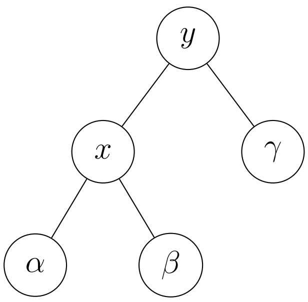
6.2.2 RB-Insert e RB-Delete

A differenza di quelle citate precedentemente, che risultano semplici sia come complessità asintotica che come implementazione, Bisogna porre particolare attenzione a queste due procedure: `RB-Insert` e `RB-Delete`.

Per ovviare a ciò, posso utilizzare le **rotazioni**. Consideriamo il seguente albero, in cui x e y sono nodi normali, mentre α, β e γ sono sotto-alberi (il colore dei nodi non ha importanza ai fini della procedura che andremo a vedere)[1]:



Applichiamo la procedura `Left(T,x)`, ottenendo:



[1]Di conseguenza, applicandola a un **RB-tree**, gli assiomi di validità potrebbero venire violati.

Osservazione La **visita simmetrica** è identica per i due alberi:

$$\alpha \rightarrow x \rightarrow \beta \rightarrow y \rightarrow \gamma$$

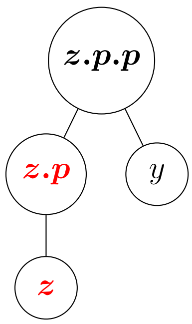
LEFT(T, x)

$x.right = y.left$
$x.right.p = x$
$y.left = x$
$x.p = y$
TRANSPLANT(T, x, y)

RB-Insert(T, z) Voglio inserire z nell'albero T. L'idea è quella di porre $z.col$ = RED poichè meno insidioso[1].

- Se violo (2) $\Rightarrow$ $z.col$ = BLACK;
- Se violo (4):
  - Risolvo localmente;
  - Sposto verso l'alto il problema.

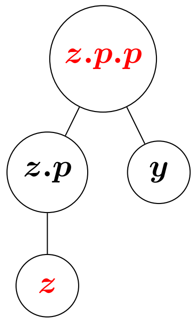
Abbiamo due **macrocasi**. $z.p$ è figlio sinistro, oppure destro. Noi analizzeremo solo il primo: **z.p figlio sinistro**.



[1]Andando a modificare il numero di nodi neri, cambia l'altezza nera, e la cosa è difficile da sistemare.

Abbiamo due possibilità per y[1]:

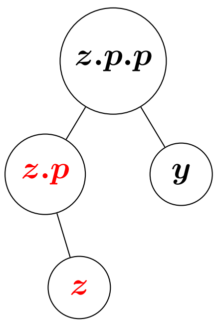
1. $y.col$ = RED. Ci è sufficiente invertire il colore di $z.p.p$ con quello dei figli.



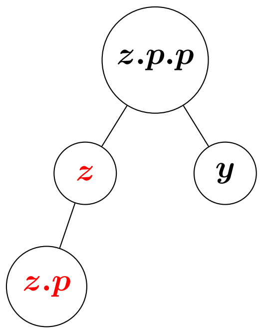
   In questo modo, risolviamo localmente e rimandiamo il problema in alto.

2. $y.col$ = BLACK. Possiamo distinguere due sottocasi.

   (2.1) z figlio destro.

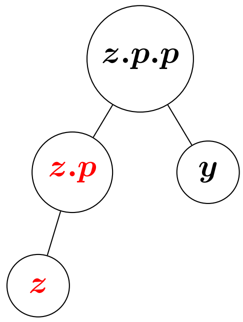


   Voglio finire nel caso (2.2). Applico `Left(T,z.p)`, ottenendo:

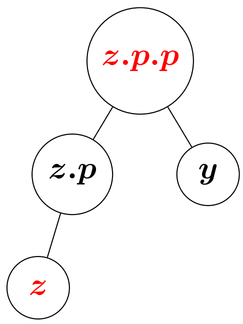


[1] I nodi con testo in rosso sono RED, quelli in grassetto sono BLACK, e quelli normali possono essere sia rossi che neri.

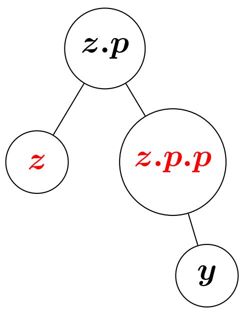
(2.2) z figlio sinistro.



Scambio i colori di $z.p.p$ con $z.p$, ottenendo:



Applico `Right(T,z.p.p)`[1]:



RB-INSERT(T, z)

INSERT(T, z)
$z.col$ = RED
RB-INSERTFIX(T, z)

[1] Analoga di `Left`.

```
RB-FixUp(T, z)
while z.p.col = RED
    if z.p = z.p.p.left // Macrocaso z.p figlio sinistro
        y = z.p.p.right
        if y.col = RED // Caso 1
            z.p.p.col = RED
            z.p.col = BLACK
            y.col = BLACK
            z = z.p.p
        else // Caso 2
            if z = z.p.right // Caso (2.1)
                Left(T, z.p)
                z = z.left
            // Caso (2.2)
            z.p.col = BLACK
            z.p.p.col = RED
            Right(T, z.p.p)
    else ... // Macrocaso z.p figlio destro
T.root.col = BLACK
```

Costo

$$\frac{h}{2} \cong \frac{\log n}{2} \approx \log n \text{ iterazioni senza rotazioni} + \textsc{Max}\ 2 \text{ rotazioni.}$$

RB-Delete(T, z) La `Delete` è ancora più problematica[1]. Se z è rosso, non ho nessun problema, poichè l'altezza nera non viene toccata. Altrimenti, i problemi possono essere diversi (radice rossa, due nodi rossi adiacenti, altezza nera inconsistente, ecc. . .).

[1] Ho deciso di ometterla per non saper come rappresentarla in modo adeguato.Darò solo una breve osservazione

6.3 Arricchimento di Strutture Dati

Vedremo due esempi, uno per gli RB-trees, e un altro per gli ABR.

- **Statistiche d'ordine** (6.3.1)
- **Interval Trees** (6.3.3)

6.3.1 Statistiche d'ordine

Struttura che parte da un RB-tree. Aggiungo:

- `Select(T,i)` $\equiv$ nodo x che occuperebbe la posizione i nei nodi ordinati per chiave (in una **visita simmetrica**);
- `Rank(T,x)` $\equiv$ posizione i (in una **visita simmetrica**) che occupa il nodo x.

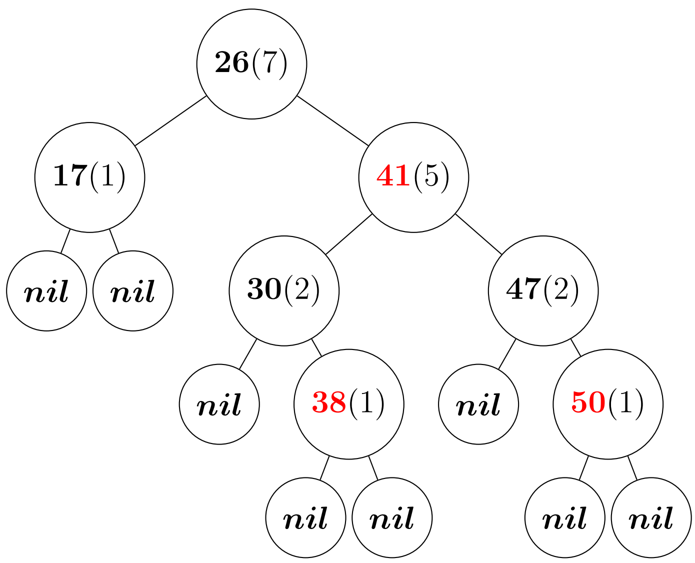
Per implementare queste due procedure, ho bisogno di un nuovo campo dati. Aggiungo il campo

$$x.size = \#\text{nodi radicati nel sottoalbero } T_x$$

Valgono

$$T.nil.size = 0$$
$$x.size = x.left.size + x.right.size + 1$$

Esempio In ogni nodo, tra le parentesi è riportato la *size* di quel nodo. Ricordiamo che i nodi *nil* (*T.nil*) hanno *size* = 0.



Vediamo un'implementazione non efficiente della procedura `Size`.

```
SIZE(x)
if x = T.nil
    x.size = 0
else
    l = SIZE(x.left)
    r = SIZE(x.right)
    x.size = l + r + 1
return x.size
```

Costo Il costo è $O(n)$, che come preannunciato, non è efficiente. Questo perchè le procedure `Insert/Delete` di un RB-tree sono nel peggiore dei casi $O(h)$.

Questa procedura, `Select`, restituisce il nodo di posizione i in T_x.

```
SELECT(x, i)
    // Pre: x ≠ T.nil, i : 1 ≤ i ≤ x.size
r = x.left.size + 1
if i = r
    return x
else if i < r
    return SELECT(x.left, i)
else
    return SELECT(x.right, i − r)
```

Costo di Select $O(h) = O(\log n)$

`Rank` restituisce la posizione i che occupa il nodo x.

```
RANK(x)
r = x.left.size + 1
y = x
while y.p ≠ T.nil // idea: r contiene la posizione di x in T_y
    if y.p.right = y
        r = r + y.p.left.size + 1
    y = y.p
return r
```

Costo di Rank $O(h) = O(\log n)$

Vediamo ora la variante di `RB-Insert`

```
RB-Insert(T, z)
    // (1) versione aggiornata di Insert
    z.size = 1
    x = T.root
    y = T.nil
    while x ≠ T.nil
        x.size = x.size + 1
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right
    z.p = y
    if y = NIL
        T.root = z
    else
        if z.key < y.key
            y.left = z
        else
            y.right = z
    // (2) RB-FixUp
    z.col = RED
    RB-FixUp(T, z)
```

E la versione aggiornata di `Left`

```
Left(T, x)
    x.right = y.left
    x.right.p = x
    y.left = x
    x.p = y
    Transplant(T, x, y)
    y.size = x.size
    x.size = x.left.size + x.right.size + 1
```

6.3.2 Teorema dell'aumento degli RB-Trees

Def. Sia $x.field$ un campo che si calcola in $O(1)$ usando $x, x.left, x.right$ ($x.field = F(x, x.left, x.right)$). Allora è possibile modificare `RB-Insert` e `RB-Delete` in modo che mantengano aggiornato il campo $x.field$ con complessità asintotica $O(\log n)$.

6.3.3 Interval Trees

Gli **Interval Trees** sono alberi binari di ricerca con un campo $x.int$, che a sua volta presenta due campi:

- $int.low$, che è anche la chiave;
- $int.high$.

E anche di un campo $x.max$ = max estremo di intervallo per i nodi in T_x, ossia

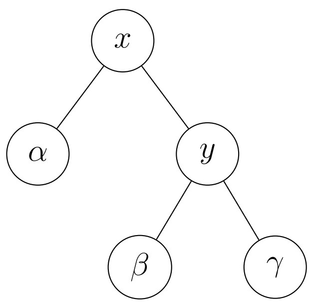
$$x.max = max \begin{cases} x.left.max \\ x.right.max \\ x.int.high \end{cases}$$

L'idea è quella in cui ogni nodo rappresenti un intervallo.

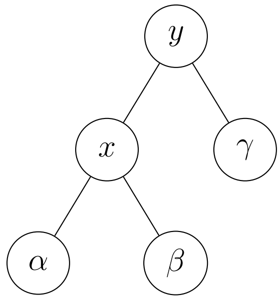
Vogliamo implementare le seguenti procedure:

- `Insert(T,x)`
- `Delete(T,x)`
- `ISearch(T,i)` con $i = [low, high]$:
  - x tale che $x.int \cap i \neq \varnothing$;
  - $T.nil$ se un tale x non c'è.

Rotazioni Prendiamo il seguente albero di esempio.



Applico `Left(T,x)`, ottenendo



Sistemo i massimi. `Left` costa ancora $O(1)$

- $y.max = x.max$
- $x.max = max\{x.int.high, x.left.max, x.right.max\}$

Vediamo `ISearch`.

ISEARCH(x, i)

```
if (x = T.nil) or (x.int ∩ i ≠ ∅)
      return x
else if (x.left) and (x.left.max ≥ i.low)
      return ISEARCH(x.left, i)
else
      return ISEARCH(x.right, i)
```

Correttezza

- **Else if**. Consideriamo in $x.left$ un intervallo i'. Abbiamo 2 possibilità.
  (1) $i \cap i' \neq \varnothing$
  (2) $i \cap i' = \varnothing$, ovvero vale $i.high < i'.low$. Questo varrà per ogni nodo dei sotto-alberi, quindi è inutile ispezionare gli antenati di quel sotto-albero.
- **Else**. $\forall\ i'$ in $x.left \Rightarrow i' \cap i \neq \varnothing$.

Costo $O(h) = O(\log n)$

Esercizio Vai a B.3

7 Lezioni del 09-10-11/05/2018

7.1 Programmazione Dinamica

Da Wikipedia:

> In Informatica la **programmazione dinamica** è una tecnica di progettazione di algoritmi basata sulla divisione del problema in sottoproblemi e sull'utilizzo di sottostrutture ottimali.

La programmazione dinamica è utilizzata in casi in cui i sottoproblemi vengono richiamati molte volte, rendendo conveniente salvare in memoria il risultato di tali sottoproblemi per non doverli ricalcolare a ogni iterazione. Ad esempio, possiamo immaginare che i dati vengano immagazzinati in una tabella:

P_1	sol. P_1	costo P_1
P_2	sol. P_2	costo P_2
...	...	...

1. Caratterizzo la "struttura" della **soluzione ottima** (soluzione ottima di P in termini di soluzioni ottime di sottoproblemi);
2. Espressione ricorsiva del **valore** della soluzione ottima;
3. Algoritmo che calcola il "valore della soluzione ottima":

$$\Downarrow$$

$$\begin{cases} \text{top down} \\ \text{bottom up} \end{cases}$$

4. Algoritmo che calcola la **soluzione** e il **valore**

7.1.1 Taglio delle aste

Introduciamo la programmazione dinamica con un primo esempio. Ipotizziamo ci sia un'azienda che produce aste di metallo molto lunghe, per venderle tagliate a pezzi.

Ogni pezzo ha un valore di vendita legato alla lunghezza. Voglio dei tagli che massimizzino il ricavo. Abbiamo

- Lunghezza complessiva dell'asta $= n$;
- Prezzi $p_1, p_2, \ldots, p_n$.
- r_n Ricavo massimo ottenibile per l'asta di lunghezza n

Esempio $n = 7$

Lunghezza	1	2	3	4	5	6	7
p_i	1	5	8	9	10	17	17

Come possiamo suddividere l'asta di lunghezza 7?

$$\begin{aligned} \text{Suddivisione} &\rightarrow p_i \\ 1+1+\cdots+1 &\rightarrow 7 \\ 2+1+\cdots+1 &\rightarrow 10 \\ 2+2+2+1 &\rightarrow 16 \\ 7 &\rightarrow 17 \\ 6+1 &\rightarrow 18 \\ 2+2+3 &\rightarrow 18 \\ &\dots \end{aligned}$$

Sono possibili 2^{n-1} possibili combinazioni di tagli. È evidente che calcolarle tutte esplicitamente risulta estremamente inefficiente.

Supponiamo di tagliare l'asta in posizione i, tale che si ottenga una soluzione con due sottoproblemi ottimali (ottenendo le sotto-aste a e b).

$$\begin{aligned} r_n &= r_i + r_{n-i} \\ r_n &= a + b \text{ con } a < r_i \\ r'_n &= r_i + b > r_n \text{ (non ha particolare senso)} \\ &i \text{ è ottimo?} \\ r_0 &= 0 \\ r_n &= max\left(\{p_n\} \cup \{r_i + r_{n-i} \mid i = 1, .., n-1\}\right) \\ &\qquad (p_n \text{ non taglio, } r_{n-1} \text{ taglio, e ottimizzo le due parti}) \\ &= max\{p_i + r_{n-i} \mid i = 1, .., n\} \end{aligned}$$

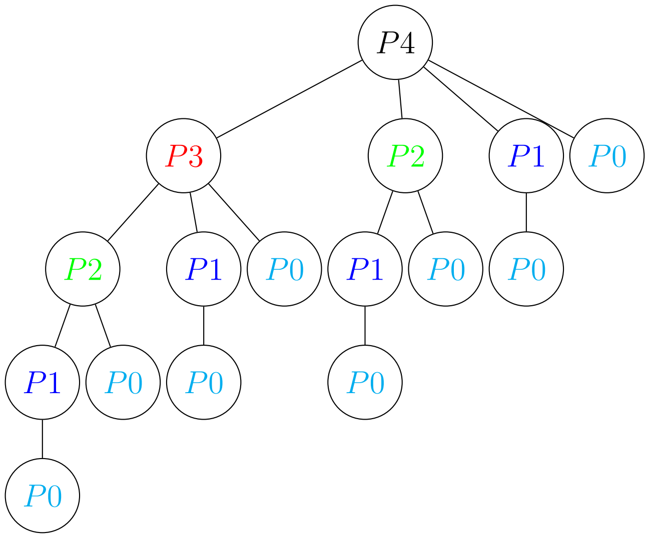
Pseudocodice

```
CutRod(p, n)
if n = 0
    return 0
else
    q = −1
    for i = 1 to n
        q = Max(q, P[i] + CutRod(P, n − i))
return q
```

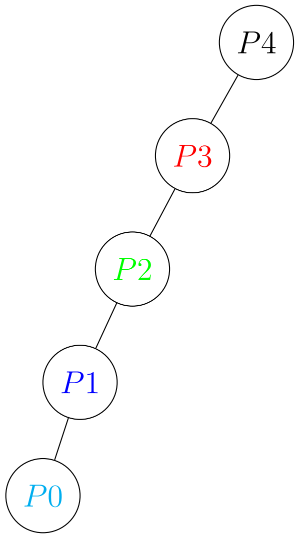
Complessità

$$
\begin{aligned}
T(n) &= 1 + \sum_{i=1}^{n} T(n-i) \\
&= 1 + \sum_{j=0}^{n-1} T(j) = \Theta(2^n) \\
T(0) &= 1 + 0 = 1 = 2^0 \\
T(n+1) &= 1 + \sum_{j=0}^{n} T(j) \\
&= 1 + \sum_{j=0}^{n-1} T(n) + T(n) \qquad \left(\sum_{j=0}^{n-1} T(n) = T(n)\right) \\
&= 2T(n)
\end{aligned}
$$

Vediamo il problema della ripetizione dei sottoproblemi con $n = 4$



Sfruttando la "memoria", restano i sottoproblemi risolti una sola volta:



- #sottoproblemi polinomiale (n^k) nella dimensione del problema di partenza;
- Esiste un algoritmo polinomiale per calcolare la soluzione del problema date le soluzioni dei sottoproblemi.

Sono possibili due approcci:

- Top-down con **memoization**;
- Bottom-up.

Top-down

```
MemCutRod(p, n)
for i = 1 to n
    r[i] = −1
return MemCutRod(p, n, r)
```

```
MemCutRod-aux(p, j, r)
if r[j] < 0
    if j = 0
        r[j] = 0
    else
        q = −1
        for i = 1 to j
            q = Max(q, p[i] + MemCutRod-aux(p, j − i, r))
            r[j] = q
return r[j]
```

$$j = 1, \ldots, n$$

$$\sum_{j=1}^{n} (\Theta(1) + j\Theta(1))$$
$$= \Theta\left(\sum_{j=1}^{n}(1+j)\right) = \Theta(n^2)$$

Bottom-Up

```
BottomUpCutRod(p, n)
alloco r[1 .. n]
r[0] = 0
for j = 1 to n // calcolo r[j]
    q = −1
    for i = 1 to j
        q = Max(q, p[i] + r[j − 1]) // Θ(1) r[j] = q
return r[n]
```

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

7.1.2 Prodotto di Matrici

Ecco un altro esempio di **programmazione dinamica**.

Consideriamo A e B di dimensioni $p \times q$ e $q \times r$ $\rightarrow$ otteniamo C di dimensioni $p \times r$.

$$C[i,j] = \sum_{k=1}^{q} A[i,k] \cdot B[k,j]$$

$$\textit{costo} = q \text{ prodotti di scalari per ogni elemento} = q \cdot p \cdot r$$

Vogliamo moltiplicare n matrici $C = A_1 \times A_2 \times \ldots \times A_n$. Possiamo farlo in molti modi poiché è un'operazione associativa e vogliamo il modo che ci permetta di fare meno prodotti scalari possibili.

Esempio

$$A_1 \times A_2 \times A_3$$

$$A_1 \rightarrow 10 \times 100$$

$$A_2 \rightarrow 100 \times 5$$

$$A_3 \rightarrow 5 \times 50$$

Abbiamo due possibili **parentetizzazioni**:

$$\begin{array}{ll} (A_1 \times A_2) \times A_3 & (10 \times 100 \times 5) + 100 \times 5 \times 50 = 7500 \\ A_1 \times (A_2 \times A_3) & 10 \times 100 \times 50 + (100 \times 5 \times 50) = 75000 \end{array}$$

Non posso provare tutte le parentetizzazioni del prodotto perché sono in numero

$$P(n) = \begin{cases} 1 & \text{se } n = 1 \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{se } n > 1 \end{cases}$$

Che cresce in maniera esponenziale! $\Omega(2^n)$

$$\begin{aligned} P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) &\geq \sum_{k=1}^{n-1} c \cdot 2^k \cdot c \cdot 2^{n-k} \\ &\geq (n-1)c^2 \cdot 2^n \geq c \cdot 2^n \text{ per } c \geq 1,\ n > 2 \end{aligned}$$

Sottoproblema $A_i \times \ldots \times A_j$, $1 \le i < j \le n$, in numero $\Theta(n^2)$

1. Suppongo di avere la *soluzione ottima*. Avrò un livello esterno di parentesizzazione

   $$(A_1 \times \ldots \times A_k)(A_{k+1} \times \ldots \times A_n)$$

   Ognuna delle due parentesizzazioni è ottima per quel sottoproblema.

   Nota bene: Ogni matrice A_i ha dimensioni $p_{i-1} \times p_i$, quindi serve un array delle dimensioni $p[0\,..\,n]$;

2. Considero i sottoproblemi $A_{ij} = (A_1 \times A_2 \times \ldots \times A_j)$

   Indico con $m[i,j]$ il valore della **soluzione ottima** per il *sottoproblema*;

   $$\begin{aligned} m[i,j] &= \text{minimo \# di prodotti per calcolare } A_{ij} \\ &= \begin{cases} 0 & i = j \\ min_{i \le n < j}\,(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i < j \end{cases} \end{aligned}$$

   È il minimo delle due parentesizzazioni + il costo del prodotto fra le due matrici ottenute.

Pseudocodice

```
MatrixChain(p, i, j)
if i = j
    return 0
else
    cmin = +∞
    for k = i to j − 1
        q = MatrixChain(p, i, k) + MatrixChain(p, k + 1, j)+
          +p[i − 1]p[k]p[j]
        if q < cmin
            cmin = q
return cmin
```

Complessità Mi aspetto poca efficienza.

$$\begin{aligned} T(n) &= 1 + \sum_{k=1}^{n-1} (T(k)T(n-k) + 1) && \text{[1 costante qualsiasi]} \\ &= n + 2\sum_{j=1}^{n-1} T(j) && \text{[trovo } T(k) \text{ due volte]} \\ &= \Omega(2^n) \end{aligned}$$

Dimostrazione per induzione (dimostrazione che $T(n) \geq 2 \; \forall \; n \geq 1$)

- **Base**: $T(1) = 1 \geq 2^{1-1}$ ok;
- **Induzione**:

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

Anche in questo problema, i sottoproblemi vengono risolti molte volte. Ecco perché è possibile un approccio alla programmazione dinamica per migliorare notevolmente l'efficienza.

- **Soluzione Top-down**: memorizzo in $m[i, j]$ la soluzione del sottopb i, j.

```
MatrixChain(p, i, j)
for i = 1 to n
    for j = 1 to n
        m[i, j] = +∞
return MatrixChainRec(p, 1, n, m)
```

```
MatrixChainRec(p, i, j, m)
if m[i, j] = +∞
    if i = j
        m[i, j] = 0
    else
        for k = 1 to j − 1
            q = MatrixChainRec(p, i, k, m)+
              +MatrixChainRec(p, k + 1, j, m)+
              +p[i − 1]p[k]p[j]
            if q < m[i, j]
                m[i, j] = q
return m[i, j]
```

Per ogni coppia (i, j) ho un ciclo di costo $O(n)$ e ho n^2 coppie

$$\Rightarrow \text{complessità } O(n^3)$$

- **Soluzione Bottom-up**

  per lunghezza <u>crescente</u> della sequenza $A_i \mathinner{\ldotp\ldotp} A_j$

  MatrixChain(p, n)

```
for i = 1 to n
      m[i, j] = 0
for l = 2 to n
      for i = 1 to n − l + 1
            j = i + l − 1
            m[i, j] = +∞
            for k = 1 to j − 1
                  q = m[i, j] + m[k + 1, j] + p[i − 1]p[k]p[j]
                  if q < m[i, j]
                        m[i, j] = q
return m[1, n]
```

  Complessità $O(n^3)$

$$\sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{j-1} 1 = \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (l-1) =$$

$$\sum_{l=2}^{n} (n-l+1)(l-1) = (n-(l-1)) \qquad (h = l-1)$$

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

$$= n\frac{n(n-1)}{2} - \frac{(n-1)n(2n-1)}{6} =$$

$$= n(n-1)\left(3\frac{n}{6} - \frac{2n-1}{6}\right) = \frac{n(n-1)(n-2)}{6} = \frac{n^3-n}{6} = \Theta(n^3)$$

Stampa del risultato

$$S[i,j] = \text{posizione della parentetizzazione più esterna per } A_i \mathinner{\ldotp\ldotp} A_j$$

```
PrintParen(S, i, j)
if i = j
    Print "A_i"
else
    Print "("
    PrintParen(S, i, S[i, j])
    PrintParen(S, S[i, j] + 1, j)
    Print ")"
```

7.1.3 Cammino minimo di un grafo*

Esempio omesso.

8 Lezioni del 16/05/2018

8.1 Altri esempi di progr. dinamica

8.1.1 Longest Common Subsequence (LCS)

Consideriamo le **basi azotate** del DNA: *Adenina, Citosina, Guanina, Timina* (A, C, G, T).

Date due stringhe

$$x_1 : \quad A\ C\ T\ A\ C\ C\ T\ G$$
$$x_2 : \quad A\ T\ C\ A\ C\ C$$

Definiamo

eliot distance: n passi per rendere uguali le due stringhe.

LCS tra x_1 e x_2: $A\ T\ A\ C\ C$

Consideriamo $X = x_1 \dots x_m$,
e una sua sottosequenza $x_{i_1} \dots x_{i_k} \quad$ con $i_1, \dots, i_k \in \{1, \dots, m\}$

Problema Date due sequenze

$$X = x_1 \dots x_m$$
$$Y = y_1 \dots y_n$$

Trovare $W = w_1 \dots w_k$ tale che:

1. W è sottosequenza di X e Y;
2. la lunghezza sia massima.

Osservazioni

1. C'è una sottosequenza comune;
2. LCS non è unica (e.g. consideriamo le stringhe AB e BA. La loro LCS è sia A che B)
$$LCS(X, Y) = \text{insieme di LCS di } X \text{ e } Y$$
3. Ricerca esaustiva impossibile.

Sottostruttura ottima Mi riduco a prefissi

$$X = x_1 \dots x_m$$

Prefissi $X^k = x_1 \dots x_k \quad k \leq m$ (ad esempio X^0 è ε)

Sia $W = w_1 \dots w_k \in LCS(X, Y)$
Allora

1. Se $x_m = y_n$ allora $w_k = x_m = y_n$ e $W^{k-1} \in LCS(X^{m-1}, Y^{n-1})$;
2. Se $x_m \neq y_n$
   (2a) se $x_m \neq w_k$ allora $W \in LCS(X^{m-1}, Y)$;
   (2b) se $y_n \neq w_k$ allora $W \in LCS(X, Y^{n-1})$.

Dimostrazione

1. $$X = x_1 \dots x_{m-1} x_m$$
   $$Y = y_1 \dots y_{n-1} y_n$$
   - $w_k = x_m = y_n$
     $$\text{Se } w_k \neq x_m \Rightarrow W \text{ sottoseq. di } X^{m-1}, Y^{n-1}$$
     $$\text{Impossibile, } Wx_m \text{ sarebbe sottoseq. di } X \text{ e } Y \text{ più lunga di } W$$
   - $W^{k-1} \in LCS(X^{m-1}, Y^{n-1})$
     W^{k-1} se non ottima, esisterebbe W' sottoseq. di X^{m-1} e Y^{n-1} tale che $|W'| > \left|W^{k-1}\right|$, ottenendo $W'x_m$ sottoseq. di X e Y con $|W'x_m| > \left|W^{k-1}x_m\right| = |W|$, **assurdo**.
2. $x_m \neq y_n$
   (2a) $x_m \neq w_k$
   $$\Rightarrow W \text{ è sottoseq. di } X^{m-1}$$
   $$W \text{ è sottoseq. di } Y$$
   $W \in LCS(X^{m-1}, Y)$? (è ottima?)
   Se <u>no</u>, esisterebbe W' sottoseq. di X^{m-1} e Y
   $$|W'| > |W|$$
   Assurdo, dato che W' è anche sottoseq. di X e Y, inoltre W è ottima per X e Y;
   (2b) Duale.

Espressione ricorsiva Valore della soluzione ottima

Dati

$$X = x_1 \dots x_m$$

$$Y = y_1 \dots y_n$$

$$C[i,j] = \text{lunghezza di LCS di } X^i \text{ e } Y^j$$

$$C[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ C[i-1,j-1]+1 & \text{se } i,j > 0 \text{ e } x_i = y_j \\ max(C[i-1,j], C[i,j-1]) & \text{se } i,j > 0 \text{ e } x_i \neq y_j \end{cases}$$

$$b[i,j] = \begin{cases} \nwarrow & \text{se } x_i = y_j \text{ e } C[i,j] = 1 + C[i-1,j-1] \\ \leftarrow & \text{se } x_i \neq y_j \text{ e } C[i,j] = C[i-1,j] \\ \uparrow & \text{se } x_i \neq y_j \text{ e } C[i,j] = C[i,j-1] \end{cases}$$

Pseudocodice

```
LCS(X, Y)
m = X.len, n = Y.len
for i = 0 to m // Θ(m)
    C[i, 0] = 0
for j = 1 to n // Θ(n)
    C[0, j] = 0
for i = 1 to m // Θ(n · m)
    for j = 1 to n
        if X[i] = Y[j] // C'è un match
            C[i, j] = C[i − 1, j − 1]
            b[i, j] = ↖
        else
            if C[i − 1, j] ≥ C[i, j − 1]
                C[i, j] = C[i − 1, j]
                b[i, j] = ←
            else
                C[i, j] = C[i, j − 1]
                b[i, j] = ↑
return b, c
```

```
PrintLCS(X, Y)
b, C = LCS(X, Y)
PrintLCSRec(X, b, m, n)
```

```
PrintLCSRec(X, b, i, j)
if i > 0 and j > 0
    if b[i, j] = ↖
        PrintLCSRec(X, b, i − 1, j − 1)
        PrintX[i]
    else if b[i, j] = ←
        PrintLCSRec(X, b, i − 1, j)
    else
        PrintLCSRec(X, b, i, j − 1)
```

Appendices

A Raccolta algoritmi

A.1 Insertion Sort

Per approfondire, vedi la sezione 2.2

```
InsertionSort(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

```
MergeSort(A, p, r)
if p < r
    q = ⌊(p+r)/2⌋ // arrotondato per difetto
    MergeSort(A, p, q) // ordina A[p..q]
    MergeSort(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

```
INSERTIONSORT(A, j)
if j > 1
      INSERTIONSORT(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 InsertionSort(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 = ⌊(p+r)/2⌋ // 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 <u>2 casi</u>.

- 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

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

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

$V[0 \mathinner{..} key] \leftarrow \textsc{false}$ // $\Theta(key) = O(top) = O(1)$
$i = 1$
$found = \textsc{false}$
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 Heap Sort

Per approfondire, vedi 4.1.

```
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])
        max = l
    else
        max = i
    if (r ≤ A.heapsize) and (A[r] > A[max])
        max = r
    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 4.1.2)

```
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)
```

$\textsc{DeleteKey}(A, i)$

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

B.2 Esercizi svolti il 06/04/2018

Gap Abbiamo un **gap** se $i < n$ t.c. $A[i+1] - A[i] > 1$.
Mostrare per induzione che se $A[n] - A[1] \geq n$ allora c'è un gap.

Sol.

- ∘ Base
  - · $n = 1 \Rightarrow A[1] - A[1] = 0$;
  - · $n = 2 \Rightarrow A[2] - A[1] > 1$ allora 1 è un gap.
- ∘ Passo induttivo $n > 2$ Se $A[n] - A[1] \geq n$ abbiamo due casi:

1. Se $A[n] - A[n-1] > 1$ allora $A[n-1]$ è un **gap**;
2. Se $A[n] - A[n-1] \leq 1$ allora

$$A[n-1] - A[1] = (A[n] - A[1]) - (A[n] - A[n-1])$$
$$(\text{valgono } A[n-1] - A[1] \geq n \text{ e } A[n] - A[n-1] \leq 1)$$
$$A[n-1] - A[1] \geq n-1 \Rightarrow A[1 \mathinner{..} n-1] \text{ contiene un } \textbf{gap}$$

```
Gap(A, p, r)
    // pre: r − p > 1
    if p = r + 1
        return p
    else
        q = (p+r)/2
        if A[q] − A[p] > q − p + 1
            Gap(A, p, r)
        else
            Gap(A, q + 1, r)
```

Valutiamo la complessità con il **Master Theorem**.

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

Abbiamo $a = 1,\ b = 2$. Confronto $f(n) = \Theta(1)$ con $n^{\log_2 1} = 1$.

Siamo nel **secondo caso** del Master Theorem, poichè le due funzioni hanno lo stesso andamento $\Rightarrow \Theta(\log^{\log_2 1} \log n) = \Theta(\log n)$

Select `Select(A, k)`, ritorna il k-esimo elemento dell'array `A` se fosse ordinato.

Sol. Ci sono diverse soluzioni al problema.

(1) Trasformo A in un **MinHeap**, ed estraggo il minimo k volte.

```
Select(A, k)
    BuildMaxHeap(A) // Θ(n)
    for i = 1 to k
        x = ExtractMin(A) // k · O(log n)
    return x
```

Complessità: $O(n + k \log n)$

(2) Soluzione alternativa.

Select(A, k)
BuildMaxHeap(A, k) // $\Theta(k)$
for $i = k + 1$ **to** n
    if $A[i] < A[1]$ // $O((n-k)\log k)$
        $A[i] \leftrightarrow A[1]$
        MaxHeapify$(A, 1, k)$
return $A[1]$

Complessità: $O(k + (n-k)\log k) \cong O(n \log k)$

Inversioni Quante **inversioni** ci sono in A? Implementare un algoritmo **Divide ed Impera** con complessità $O(n \log n)$

Def (inversione) Una coppia di indici i, j tali che $i < j$ è un **inversione** per l'array A se e solo se $A[i] > A[j]$.

Sol. Possiamo distinguere tre casi:

- $i, j \in [p, q]$;
- $i, j \in [q+1, r]$;
- $i \in [p, q], j \in [q+1, r]$.

La soluzione che adotteremo conterà le inversioni dei tre casi separatamente, e restituirà la somma dei tre valori ottenuti.

$$\begin{aligned}\#\ \text{Inv}(p, r) = \#\ \text{Inv}(p, q) + \#\ \text{Inv}(q+1, r) \\ + |\{(i, j) : i \in [p, q], j \in [q+1, r], A[i] > A[j]\}|\end{aligned}$$

Inv(A, p, r)
// restituisce **#***inv* e ordina l'array A
if $p < r$
    $q = \frac{p+r}{2}$
    return Inv(A, p, q) + Inv$(A, q+1, r)$ + MergeInv(A, p, q, r)
else
    return 0

(a) Inv non cambia gli elementi in $A[p,q]$ e $A[q+1,r]$;

(b) Il numero di inversioni "a cavallo" non cambia;

(c) i, j inversione $A[i] > A[j]$
$\Rightarrow (i', j) \quad i' \in [i, q]$ inversione.

```
MergeInv(A, p, q, r)
n1 = q − p + 1
n2 = r − q
L[1, n1] = A[p, q]
R[1, n2] = A[q + 1, r]
L[n1 + 1] = R[n2 + 1] = ∞
i = j = 1
inv = 0
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
        inv = inv + n1 − i + 1
return inv
```

B.3 Esercizio del 03/05/2018

ABR con:

- $x.left$
- $x.right$
- $x.succ$
- $\cancel{x.p}$, non ho il padre.

`Search`, `Max` e `Min` restano invariate. Cambiano invece `Insert`, `Succ` e c'è bisogno di una procedura `Parent`.

```
INSERT(T, z)
x = T.root
y = NIL // parent
pred = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
        pred = y
if y = NIL
    T.root = z
else if z.key < y.key
    y.left = z
else
    y.right = z
if pred ≠ NIL
    z.succ = pred.succ
    pred.succ = z
else
    z.succ = y
```

Analisi di Insert Scorro con y l'albero. Chi sono $z.succ$ e il genitore di z? Distinguiamo due casi.

a) z figlio destro di y.

$$z.succ = y.succ \qquad y.succ = z$$

b) z figlio sinistro di y.

$$z.succ = y \qquad y.succ \text{ invariato}$$

Costo di Insert $O(h)$

```
SUCC(T, z)
return x.succ
```

```
Parent(T, x)
y = x
while y.right ≠ NIL
      y = y.right
z = z.succ
if z = NIL
      w = T.root
else
      w = z.left
if w = x
      return z
while w.right ≠ x
      w = w.right
return w
```

Costo di Parent $O(h)$.

```
Pred(T, z)
if x.left ≠ NIL
      return Max(x.left)
else
      Parent(T, x)
      while (p ≠ NIL) and (x = p.left)
            x = p
            p = Parent(T, x)
      return p
```

Costo di Pred Essendoci un ciclo `while` che itera fino a h volte, e in questo ciclo viene chiamata una procedura con complessità $O(h)$, abbiamo che `Pred` ha complessità $O(h^2)$.