



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>

23 maggio 2019

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Problem Solving . . . . .	1
1.2	Analisi . . . . .	2
<b>2</b>	<b>Algoritmi di Ordinamento</b>	<b>3</b>
2.1	Problema dell'Ordinamento (Sorting) . . . . .	3
2.2	Insertion Sort . . . . .	3
2.2.1	Invarianti e Correttezza . . . . .	5
2.2.2	Complessità di Insertion Sort . . . . .	6
2.3	Divide et Impera . . . . .	8
2.4	Merge Sort . . . . .	8
2.4.1	Approfondimento sull'Induzione . . . . .	12
2.4.2	Complessità di Merge Sort . . . . .	12
2.5	Confronto tra Insertion Sort e Merge Sort . . . . .	15
<b>3</b>	<b>Limiti Asintotici e Ricorrenze</b>	<b>16</b>
3.1	Notazione Asintotica . . . . .	16
3.1.1	Limite Asintotico Superiore . . . . .	17
3.1.2	Limite Asintotico Inferiore . . . . .	19
3.1.3	Limite Asintotico Stretto . . . . .	20
3.2	Metodo del Limite . . . . .	21
3.3	Proprietà Generali . . . . .	22
3.4	Complessità di un Problema . . . . .	23
3.4.1	Esempio: limite inferiore per ordinamento basato su scambi di elementi contigui . . . . .	23
3.5	Soluzione di Ricorrenze . . . . .	24
3.5.1	Metodo di Sostituzione . . . . .	24
3.5.2	Master Theorem . . . . .	30
<b>4</b>	<b>Algoritmi di Ordinamento (cont.)</b>	<b>35</b>
4.1	Heap Sort . . . . .	35
4.1.1	Max Heap . . . . .	37
4.1.2	Code con Priorità . . . . .	40
4.2	Quick Sort . . . . .	43
4.2.1	Correttezza di QuickSort(A, p, r) . . . . .	44
4.2.2	Complessità di Quick Sort . . . . .	45
4.3	Quick Sort a Tre Partizioni . . . . .	47
4.4	Limite Inferiore . . . . .	48
4.5	Albero di Decisione . . . . .	48

4.6	Counting Sort . . . . .	50
4.6.1	Proprietà di Stabilità . . . . .	51
4.7	Radix Sort . . . . .	52
<b>5</b>	<b>Tabelle Hash</b>	<b>53</b>
5.1	Chaining . . . . .	55
5.1.1	Hashing uniforme semplice . . . . .	55
5.1.2	Funzioni Hash . . . . .	57
5.1.3	Hashing Universale . . . . .	58
5.2	Open Addressing . . . . .	59
5.2.1	Hashing uniforme . . . . .	59
5.2.2	Funzioni di Hash . . . . .	60
<b>6</b>	<b>Alberi di Ricerca</b>	<b>64</b>
6.1	Alberi Binari di Ricerca (ABR) . . . . .	64
6.1.1	Visita simmetrica . . . . .	65
6.1.2	Ricerca . . . . .	65
6.1.3	Successore di un nodo . . . . .	67
6.1.4	Inserimento . . . . .	68
6.1.5	Eliminazione di un nodo . . . . .	68
6.2	Red-Black Trees . . . . .	70
6.2.1	Complessità algoritmi RB-Trees . . . . .	72
6.2.2	RB-Insert e RB-Delete . . . . .	72
6.3	Arricchimento di Strutture Dati . . . . .	77
6.3.1	Statistiche d'ordine . . . . .	77
6.3.2	Teorema dell'aumento degli RB-Trees . . . . .	80
6.3.3	Interval Trees . . . . .	80
<b>7</b>	<b>Lezioni del 09-10-11/05/2018</b>	<b>82</b>
7.1	Programmazione Dinamica . . . . .	82
7.1.1	Taglio delle aste . . . . .	82
7.1.2	Prodotto di Matrici . . . . .	87
7.1.3	Cammino minimo di un grafo* . . . . .	91
<b>8</b>	<b>Lezioni del 16/05/2018</b>	<b>92</b>
8.1	Altri esempi di progr. dinamica . . . . .	92
8.1.1	Longest Common Subsequence (LCS) . . . . .	92
	<b>Appendices</b>	<b>96</b>

---

<b>A</b>	<b>Raccolta algoritmi</b>	<b>96</b>
A.1	Insertion Sort . . . . .	96
A.2	Merge Sort . . . . .	96
A.3	Insertion Sort ricorsivo . . . . .	97
A.3.1	Correttezza di InsertionSort(A, j) . . . . .	97
A.3.2	Correttezza di Insert(A, j) . . . . .	98
A.4	CheckDup . . . . .	98
A.4.1	Correttezza di DMerge(A,p,q,r) . . . . .	99
A.5	SumKey . . . . .	99
A.5.1	Correttezza di Sum(A, key) . . . . .	100
A.6	Heap Sort . . . . .	102
A.7	Code con Priorità . . . . .	103
<b>B</b>	<b>Esercizi</b>	<b>105</b>
B.1	Ricorrenze . . . . .	105
B.2	Esercizi svolti il 06/04/2018 . . . . .	105
B.3	Esercizio del 03/05/2018 . . . . .	108

# 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  $\rightarrow$  **Algoritmo**  $\rightarrow$  Output

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

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

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

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

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

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

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

---

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

## 1.2 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:

- InsertionSort;
- MergeSort.

### 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 inverte con l'elemento alla sua sinistra, come segue

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

 Key: 

8
---

La key analizzata è 8.

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

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

 Key: 

4
---

La key analizzata è 4.

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

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

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

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

      Key: 

7
---

Key analizzata 7.

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

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

Ottengo l'array ordinato:

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



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

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

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

**Pseudocodice** Segue lo pseudocodice dell'InsertionSort.

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       $A[i + 1] = key$ 
```

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  $n+1$ .

---

<sup>1</sup>**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       $key = A[j]$  //  $A[1 \dots j-1]$  ordinato
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i+1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i+1] = key$ 
```

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

Vediamo il **costo** di ogni istruzione:

$$c_0 \rightarrow 1$$

$$c_1 \rightarrow 1$$

$$c_2 \rightarrow n$$

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

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

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

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

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

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

---

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

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

Andiamo ad analizzare i 3 possibili casi:

- a) Caso migliore (2.2.2)
- 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:

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

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

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

La **complessità** diventa:

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

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

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

Otteniamo:

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

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

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

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

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

Comunque, l'idea è la seguente:

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

## 2.3 Divide et Impera

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

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

**impera** risolve i sottoproblemi:

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

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

## 2.4 Merge Sort

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

---

<sup>1</sup>Si consiglia di dare uno sguardo all'algoritmo anche da altre fonti, poichè presentarlo graficamente in  $\text{\LaTeX}$ , 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<sup>1</sup>:

---

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

- $L[1..4] = A[1..4]$ : indice  $i = 1$  per scorrerlo;
  - $R[1..4] = A[5..8]$ : indice  $j = 1$  per scorrerlo;
- Valuto  $L[i]$  e  $R[j]$ .
- Se  $L[i] \leq R[j]$ , inserisco  $L[i]$  e incremento  $i$ .
  - Altrimenti, inserisco  $R[j]$  e incremento  $j$ .
  - Itero finchè entrambi gli indici non sono out of bounds.

**Pseudocodice** Segue lo pseudocodice del **MergeSort**.

**MERGESORT**( $A, p, r$ )

```

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

```

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

```

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

```

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

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

**Dimostrazione per induzione su  $r-p$** 

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

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

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

Per ipotesi induttiva:

- 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

- $[P(0)]$  (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<sup>1</sup>

**Merge(A, p, q, r)**

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

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

Sommandoli, ottengo una complessità all'incirca di:

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

Nel dettaglio:

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

$\Downarrow$

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

con

---

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



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

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

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

$$\begin{array}{ccccccc}
& & & & T^{MS}(n) & & \\
& & & & an + b & & \\
& & & & & & \\
& & T^{MS}(n_1) & & T^{MS}(n_2) & & \\
& & an_1 + b & & an_2 + b & & \\
& & & & & & \\
& T^{MS}(n_{11}) & T^{MS}(n_{12}) & T^{MS}(n_{21}) & T^{MS}(n_{22}) & & \\
& an_{11} + b & an_{12} + b & an_{21} + b & an_{22} + b & & \\
& & & & & & \\
& & & \dots & & & \\
& & & & & & \\
& c_0 & c_0 & \dots & \dots & \dots & c_0 & c_0
\end{array}$$

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

$$\begin{array}{ll}
\mathbf{i} = \mathbf{0} & an + b \\
\mathbf{i} = \mathbf{1} & a(n_1 + n_2) + 2b \approx an + 2b \\
\mathbf{i} = \mathbf{2} & a(n_{11} + n_{12} + n_{21} + n_{22}) + 4b \approx an + 4b \\
& \dots \\
\mathbf{i} = \mathbf{h} & c_0 n
\end{array}$$

Poniamo  $n = 2^h$ . Abbiamo

$$\begin{aligned}
T^{MS}(n) &= \sum_{i=0}^{h-1} (an + 2^i b) + c_0 n \\
&= anh + b \sum_{i=0}^{h-1} 2^i & (h = \log_2 n) \\
&= an \log_2 n + b2^h - b + c_0 n & (2^h = n) \\
&= an \log_2 n + (b + c_0)n - b \\
T^{MS}(n) &= an \log_2 n + b''n + c'' \approx n \log_2 n
\end{aligned}$$

## 2.5 Confronto tra Insertion Sort e Merge Sort

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

Posso calcolare il limite del rapporto:

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

Per definizione

$$\forall \varepsilon > 0 \exists n_0 : \forall n \geq n_0 \quad \frac{T^{MS}(n)}{T^{IS}(n)} < \varepsilon$$

$\Downarrow$

$$T^{MS}(n) < \varepsilon T^{IS}(n) = \frac{T^{IS}}{m} \quad (\text{Ponendo, ad esempio, } \varepsilon = \frac{1}{m})$$

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

$n$	$T^{IS}(n) = n^2$	$T^{MS}(n) = n \log n$
10	0.1ns	0.033ns
1000	1ms	10μs
10 <sup>6</sup>	17 minuti	20ms
10 <sup>9</sup>	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**:

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

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

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

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

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

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

### 3.1.1 Limite Asintotico Superiore

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

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

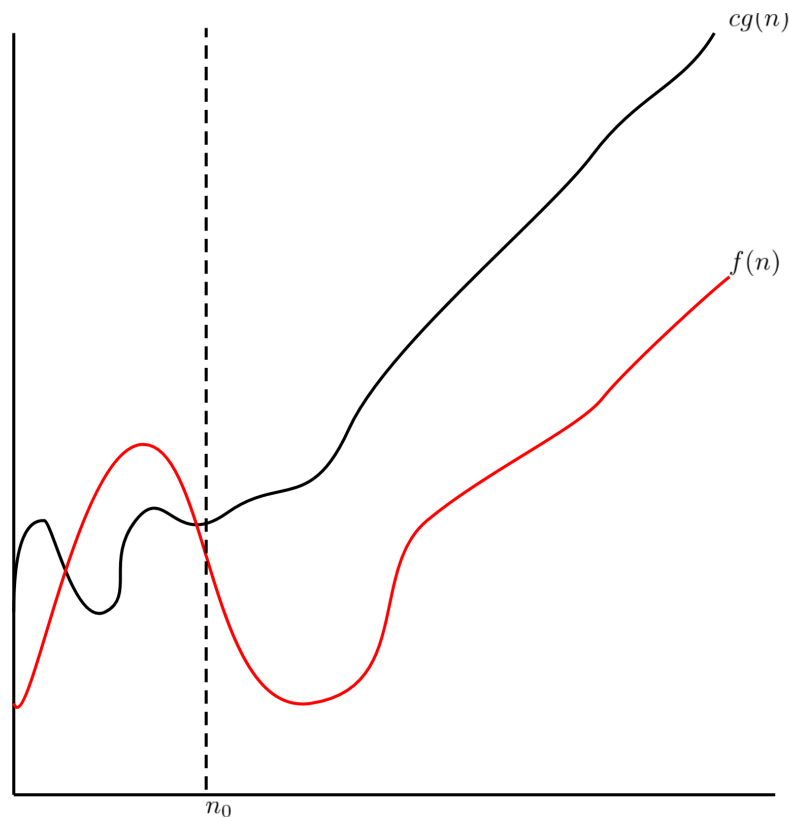


Figura 2: Rappresentazione del limite asintotico superiore per  $f(n)$

#### Esempi

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

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

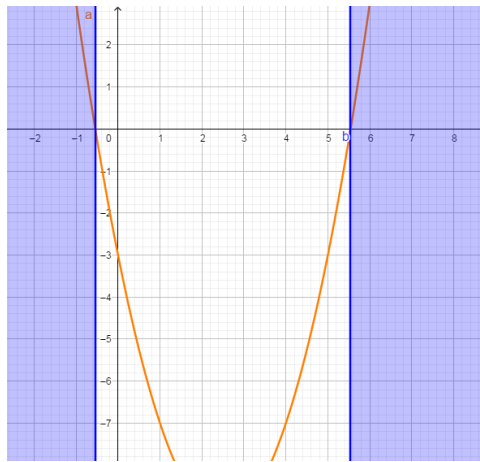
Ipotizziamo  $c = 3$

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

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

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

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



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

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

○  $f_1(n) = O(g(n^3))$  ? Sì.

$$c = 3$$

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

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

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

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

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

Vale la seguente

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

ok per  $c = 3, n_0 = 0$

### 3.1.2 Limite Asintotico Inferiore

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

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

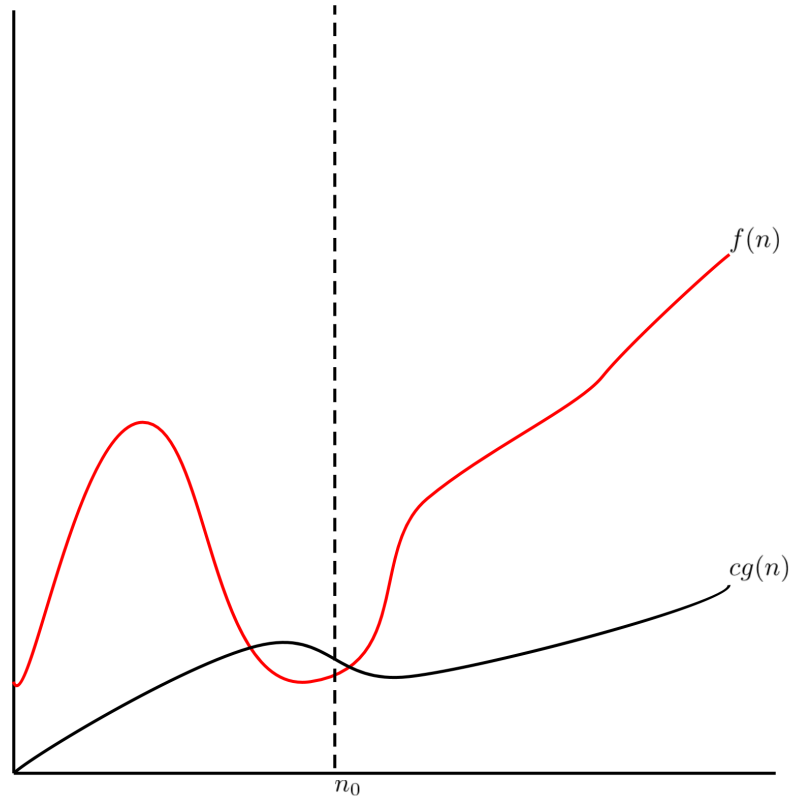


Figura 3: Rappresentazione del limite asintotico inferiore per  $f(n)$

#### Esempi

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

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

### 3.1.3 Limite Asintotico Stretto

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

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

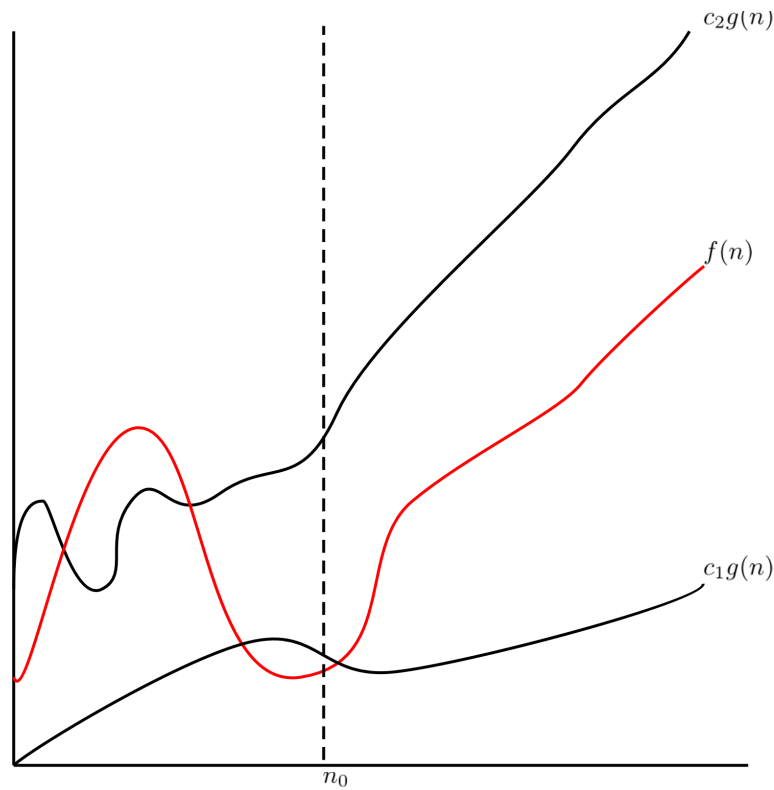


Figura 4: Rappresentazione del limite asintotico stretto per  $f(n)$

#### Esempi

$$f_1(n) = 2n^2 + 5n + 3 = \Theta(n^2)$$

$$c_1 = 1 \quad c_2 = 3 \quad n_0 = 6$$

$$f_2(n) = 2 + \sin(n) = \Theta(1)$$

$$c_1 = 1 \quad c_2 = 3 \quad n_0 = 0$$

$$f_1(n) \neq \Theta(n^3)$$

$$f_1(n) = O(n^3)$$

$$f_1(n) \neq \Omega(n^3)$$

$\Downarrow$

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



### 3.2 Metodo del Limite

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

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

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

#### Dimostrazione

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

$$\Rightarrow \forall \varepsilon > 0 \quad \exists n_0 : \forall n \geq n_0 \quad \left| \frac{f(n)}{g(n)} - k \right| \leq \varepsilon$$

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

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

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

$$\Rightarrow f(n) = \Theta(g(n))$$

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

$$\Rightarrow \forall \varepsilon > 0 \quad \exists n_0 : \forall n \geq n_0 \quad \frac{f(n)}{g(n)} \leq \varepsilon$$

$$\Rightarrow f(n) \leq \varepsilon g(n)$$

$$\Rightarrow f(n) = O(g(n))$$

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

$$\Rightarrow \exists c > 0 \quad \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \quad cg(n) \leq f(n)$$

$$\Rightarrow \text{Impossibile, infatti sia } \varepsilon = \frac{c}{2} > 0$$

$$\Rightarrow \exists n_1 \in \mathbb{N} : \forall n \geq n_1 \quad f(n) \leq \frac{c}{2}g(n) < cg(n)$$

$$\Rightarrow \text{Contraddizione!}$$

3. Si dimostra in modo analogo al punto (2)

### 3.3 Proprietà Generali

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

In generale

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

### 3.4 Complessità di un Problema

Dato un problema  $P \{ \text{INPUT} \rightarrow \text{OUTPUT} \}$ , 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  $O(f(n))$ , allora  $P$  è  $O(f(n))$ .

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

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

#### 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  $k+1$ ).

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

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

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

Ossia,  $\#inv(A)$  è massimizzato.

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 MergeSort la complessità nel modo seguente:

MERGE\_SORT( $A, p, r$ )

```

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

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

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

- **Metodo di sostituzione** (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.

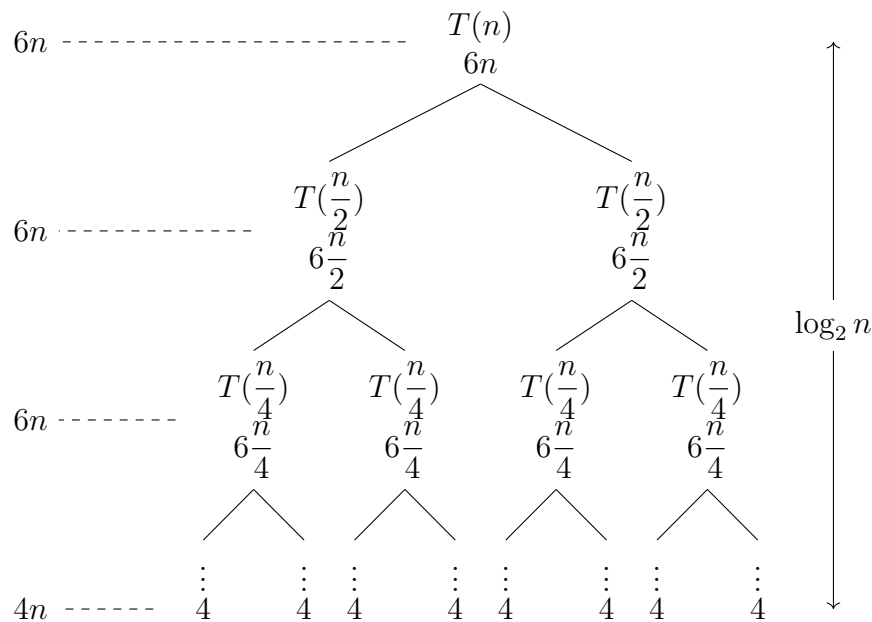
**Esempio**

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

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:



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.

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

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  $T(n)$

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

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

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

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

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

**Esercizio (importante)**

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

Voglio dimostrare che

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

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

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

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

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

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

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

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

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

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

Ipotizzo un andamento simile a MergeSort:  $\Theta(n \log n)$ . 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 ...

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



**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 ...

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

### 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

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

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

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

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

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

allora

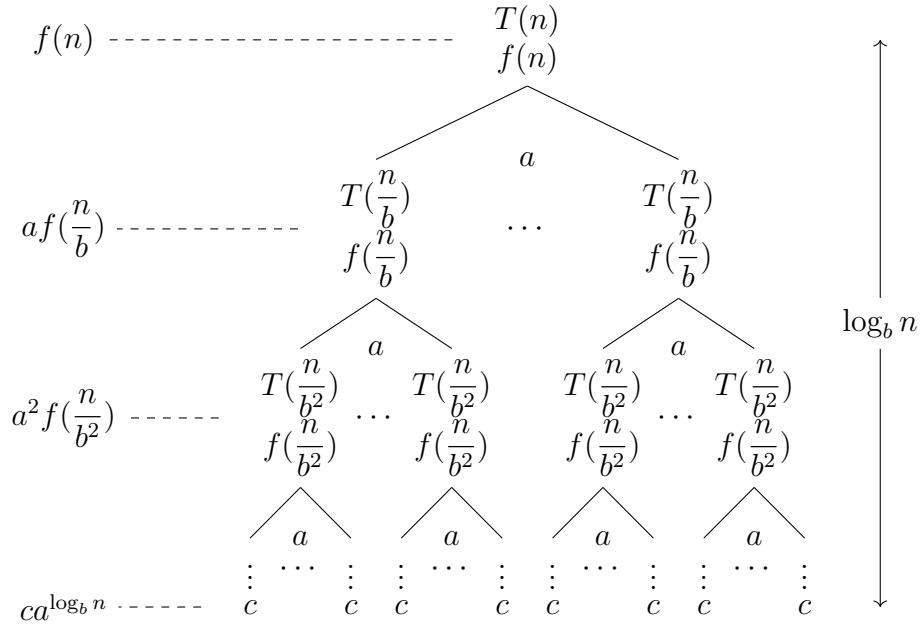
$$T(n) = \Theta(f(n))$$

**Intuizione sul perchè  $n^{\log_b a}$**

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

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

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



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

A)

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

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

B)

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

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

C)

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a}} = \infty \quad &\& \exists \varepsilon > 0 : \lim_{n \rightarrow \infty} \frac{f(n)}{n^{\log_b a + \varepsilon}} = \infty \\ &\& \text{Regolarità} \Rightarrow \text{Caso 3: } T(n) = \Theta(f(n)) \end{aligned}$$

**Esercizi**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- $T(n) = 5T\left(\frac{n}{2}\right) + n^3 \log n$

Abbiamo

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

$$f(n) = n^3 \log n \quad n^{\log_2 5} \quad (\log_2 5 < 3)$$

$$0 < \varepsilon < 3 - \log_2 5 \Rightarrow \lim_{n \rightarrow \infty} \frac{n^3 \log n}{n^{\log_2 5 + \varepsilon}} = \infty$$

Possibile **caso 3. Regolarità?**

$$af\left(\frac{n}{b}\right) \leq kf(n) \quad \text{per } 0 < k < 1 \text{ opportuno}$$

$$5\left(\frac{n}{2}\right)^3 \log \frac{n}{2} = \frac{5}{8}n^3 \log \frac{n}{2} \leq \frac{5}{8}n^3 \log n \leq kn^3 \log n \quad \text{per } 0 < k \leq \frac{5}{8} < 1$$

↓

$$\text{Caso 3: } T(n) = \Theta(f(n)) = \Theta(n^3 \log n)$$

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

$$f(n) = n^3 \log n \quad n^{\log_3 27} \quad (\log_3 27 = 3)$$

$$\lim_{n \rightarrow \infty} \frac{n^3 \log n}{n^{3+\varepsilon}} = +\infty \quad \forall \varepsilon > 0, \text{ non possiamo dimostrare } 3$$

$\Rightarrow$  Non siamo in **nessun** caso del Master Theorem.

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

$$27\left(\frac{n}{3}\right)^3 \log \frac{n}{3} = n^3(\log n - \log 3) \not< kn^3 \log n \text{ per nessun } k > 0$$

$$\text{Infatti } \frac{(\log n - \log 3)n^3}{n^3 \log n} \rightarrow 1$$

(Posso usare il Metodo di Sostituzione)

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

Costruiamo l'albero delle ricorrenze:

- **radice:** costo  $n^3 \log n$ ;
- ogni nodo ha 27 figli.
  - ◊ i 27 figli del primo livello hanno costo  $\left(\frac{n}{3}\right)^3 \log \frac{n}{3}$ ;
  - ◊ i  $27^2$  figli del secondo livello hanno costo  $\left(\frac{n}{9}\right)^3 \log \frac{n}{9}$ ;
  - ◊ ...
  - ◊ le  $27^n$  foglie terminali hanno costo  $O(1)$ .

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

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

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

Devo dimostrare che valgano le seguenti condizioni:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\exists d > 0 : T(n) \geq dn^3(\log n)^2$$

$$\geq 27\left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 + n^3 \log n$$

$$= \dots = dn^3(\log n)^2 - n^3\left(\log n(2d \log 3 - 1) - d(\log 3)^2\right)$$

$$\geq dn^3(\log n)^2$$

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

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

## 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:

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

### Memoria

- InsertionSort:

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

- MergeSort: 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'**Heap Sort**<sup>1</sup> è un algoritmo di ordinamento basato su una struttura chiamata **heap**, che prende le caratteristiche positive di InsertionSort e MergeSort:

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

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

Seguono alcune definizioni.

---

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

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

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

**Albero quasi completo:** è un albero completo a tutti i livelli eccetto l'ultimo, in cui possono mancare delle foglie 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  $i$ 
1 return  $2 * i$ 
```

**RIGHT( $i$ )**

```
// restituisce il figlio dx del nodo  $i$ 
1 return  $2 * i + 1$ 
```

**PARENT( $i$ )**

```
// restituisce il genitore del nodo  $i$ 
1 return  $\lfloor i/2 \rfloor$ 
```



### 4.1.1 Max Heap

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

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

Equivalentemente

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

#### Osservazioni

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

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

MAXHEAPIFY( $A, i$ )

```

1   $l = \mathbf{LEFT}(i)$ 
2   $r = \mathbf{RIGHT}(i)$ 
3  if ( $l \leq A.heapsize$ ) and ( $A[l] > A[i]$ )
4       $max = l$ 
5  else
6       $max = i$ 
7  if ( $r \leq A.heapsize$ ) and ( $A[r] > A[max]$ )
8       $max = r$ 
9  if ( $max \neq i$ )
10      $A[i] \leftrightarrow A[max]$ 
11     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:**  $max \neq i$ , distunguo 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  $i$

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

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

Quali sono i nodi foglia?

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

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

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

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

**BUILDMAXHEAP( $A$ )**

```

1   $A.heapsize = A.length$ 
2  for  $i = \lfloor A.length/2 \rfloor$  down to 1
3      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.

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

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

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

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

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

**HEAPSORT**(A)

```

1  BUILDMAXHEAP(A) // O(n)
2  for i = A.length down to 2
3      A[1] ↔ A[i]
4      A.heapsize = A.heapsize - 1
5      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( $S, x, \delta$ )`
- `ChangeKey( $S, x, \delta$ )`
- `Delete( $S, x$ )`

**Idea** Uso un **Max Heap** ( $A$ ).

`MAX( $A$ )`

```
1  if  $A.heapsize = 0$ 
2      error
3  else return  $A[1]$ 
```

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

`EXTRACTMAX( $A$ )`

```
1   $max = A[1]$ 
2   $A[1] = A[A.heapsize]$ 
3   $A.heapsize = A.heapsize - 1$ 
4  MAXHEAPIFY( $A, 1$ ) // ripristina le proprietà di MaxHeap
5  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:

$$\begin{aligned} &\text{Per ogni } j \neq i \\ &A[j] \leq \text{antenati} \end{aligned}$$

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

```

MAXHEAPIFYUP( $A, i$ )
1  if ( $i > 1$ ) and ( $A[i] > A[\text{PARENT}(i)]$ )
2       $A[i] \leftrightarrow A[\text{PARENT}(i)]$ 
3      MAXHEAPIFYUP( $A, \text{PARENT}(i)$ )

```

### Correttezza di MaxHeapifyUp

#### Casi base

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

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

#### Induzione

( $A[i] > A[\text{Parent}(i)]$ ) scambio le due celle. I discendenti (sottoalberi) 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( $A, x$ )
1   $A.\text{heapsize} = A.\text{heapsize} + 1$ 
2   $A[A.\text{heapsize}] = x$ 
3  MAXHEAPIFYUP( $A, A.\text{heapsize}$ )

```

**Insert** ha complessità  $O(\log n)$ , la stessa di **MaxHeapifyUp**.

```

INCREASEKEY( $A, i, \delta$ )
  // Precondizione:  $\delta \geq 0$ 
1   $A[i] = A[i] + \delta$ 
2  MAXHEAPIFYUP( $A, i$ )

```

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

```

CHANGEKEY( $A, i, \delta$ )
1   $A[i] = A[i] + \delta$ 
2  if  $\delta > 0$ 
3      MAXHEAPIFYUP( $A, i$ )
4  else //  $\delta \leq 0$ 
5      MAXHEAPIFY( $A, i$ )

```

**ChangeKey** è come **IncreaseKey**, ma può utilizzare valori di  $\delta$  qualsiasi, ed è corretto per la seguente proprietà:

Se per ogni  $j \neq i$   $A[j] \geq$  discendenti  
 $\Rightarrow$  dopo **MaxHeapify** ho un **MaxHeap**

**DELETEKEY**( $A, i$ )

```
1  old =  $A[i]$ 
2   $A[i] = A[A.heapsize]$ 
3   $A.heapsize = A.heapsize - 1$ 
4  if  $old \leq A[i]$ 
5      MAXHEAPIFYUP( $A, i$ )
6  else
7      MAXHEAPIFY( $A, i$ )
```

**DeleteKey** ha complessità  $O(\log n)$ .

## 4.2 Quick Sort

Il **Quick Sort** 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*
  - Sceglie 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( $A, p, r$ )

```

1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q$ )
4      QUICKSORT( $A, q + 1, r$ )

```

PARTITION( $A, p, r$ )

```

1   $x = A[r]$  // pivot  $A[r]$ 
2   $i = p - 1$ 
   //  $A[p, i] \leq x$ 
   //  $A[i+1, j-1] > x$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6           $A[i] \leftrightarrow A[j]$ 
7   $A[i + 1] \leftrightarrow A[r]$ 
8  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

$$\boxed{\leq A[q]} \quad \boxed{A[q]} \quad \boxed{\geq A[q]}$$

$$\text{QuickSort}(A, p, q) \quad \boxed{\leq A[q], \text{ord}} \quad \boxed{A[q]} \quad \boxed{> A[q]}$$

$$\text{QuickSort}(A, q+1, r) \quad \boxed{\leq A[q], \text{ord}} \quad \boxed{A[q]} \quad \boxed{> A[q], \text{ord}}$$

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

$$\boxed{9} \boxed{6} \boxed{0} \boxed{8} \boxed{4} \boxed{2} \quad \text{pivot: } \boxed{2}$$

$i$  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  $\Rightarrow i++, A[i] \leftrightarrow A[j], j++$

$$\boxed{0} \boxed{6} \boxed{9} \boxed{8} \boxed{4} \boxed{2} \quad \text{pivot: } \boxed{2}$$

$i$  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

$$\boxed{0} \boxed{2} \boxed{9} \boxed{8} \boxed{4} \boxed{6}$$

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

$$\boxed{0} \boxed{2}$$

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

$$\boxed{9} \boxed{8} \boxed{4} \boxed{6} \quad \text{pivot: } \boxed{6}$$

$i$  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 Quick Sort

Partition costa  $\Theta(n)$

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

$$q - p < n$$

#### Caso peggiore

$$T^{QS} = \Theta(n) + T^{QS}(n - 1) = \Theta(n^2) \quad (\Theta(n) = cn)$$

$T(n)$

$cn$

$cn - 1$

$cn - 2$

$\dots$

$d$

$$\begin{aligned} \sum_{j=1}^{n-1} c(n-j) + d &= \sum_{k=1}^n ck + d = \\ &= c \sum_{k=1}^n k + d \quad \left( \frac{c(n+1)n}{2} + d = \Theta(n^2) \right) \end{aligned}$$

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

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

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

$$n(2c - d) - c \geq 0 \quad \text{ok, } c > \frac{d}{2}$$

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

- $T(n) = \Omega(n^2)$  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**( $A, p, r$ )

- 1  $q = \text{RANDOM}(p, r)$
- 2  $A[q] \leftrightarrow A[r]$
- 3 **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( $A, p, r$ )

```

1   $x = A[r]$ 
2   $i = p - 1$ 
3   $k = p$ 
4   $j = r$ 
5  while  $k < j$ 
6      if  $A[k] < x$ 
7           $i = i + 1$ 
8           $A[i] \leftrightarrow A[k]$ 
9           $k = k + 1$ 
10     else if  $A[k] > x$ 
11          $j = j - 1$ 
12          $A[j] \leftrightarrow A[k]$ 
13     else
14          $k = k + 1$ 
15      $// k = j$ 
16      $A[j] \leftrightarrow A[r]$ 
17 return  $(i + 1, j)$  // restituisce una coppia di valori
```

QUICKSORT( $A, p, r$ )

```

1  if  $p < r$ 
2       $q1, q2 = \text{TRIPARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q1 - 1$ )
4      QUICKSORT( $A, q2 + 1, r$ )
```

## 4.4 Limite Inferiore

Input:  $a_1 \dots a_n$

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

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

**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 **InsertionSort** basato su scambi.

INSERTION-SORT( $A$ )

```

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

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

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

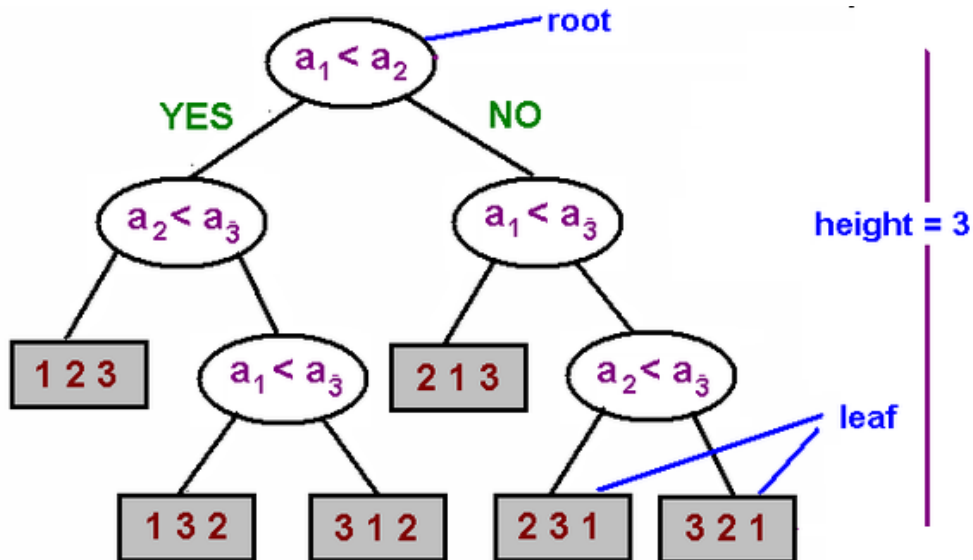


Figura 5: Albero di decisione per l'array  $A[1, 2, 3]$

### Osservazioni

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

$$\text{per IS} \quad n^2$$

$$\text{per MS} \quad n \log n$$

- Ogni foglia ha una sola permutazione. Ogni permutazione compare (almeno) in una foglia.

**In generale**, le foglie contengono tutte le permutazioni.

$$\# \text{ foglie} \geq \# \text{ permutazioni} = n! \quad (\# \text{ foglie} \leq 2^h)$$

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

Inoltre,

- # operazioni  $\geq h = \Omega(n \log n)$
- **HeapSort**, **MergeSort**  $O(n \log n)$

$\Rightarrow$  ordinamento (bastato su confronti)  $\Theta(n \log n)$

## 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:**  $A[1..n]$  con  $A[j] \in [0, k] \quad \forall 1 \leq j \leq n$ ;

**Output:**  $B[1..n]$  permutazione ordinata di  $A$ ;

**Supporto:**  $C[0..k]$ .

COUNTINGSORT( $A, B, k$ )

```

1   $C[0..k] \leftarrow 0$ 
2  for  $j = 1$  to  $A.length$ 
    //  $C[x] = \#elem$  in  $A$  con valore  $x$ 
3     $C[A[j]] = C[A[j]] + 1$ 
4  for  $i = 1$  to  $k$ 
    //  $C[x] = \#elem$  in  $A$  con valore  $\leq x$ 
5     $C[i] = C[i-1] + C[i]$ 
6  for  $j = A.length$  down to 1
7     $B[C[A[j]]] = A[j]$ 
8     $C[A[j]] = C[A[j]] - 1$ 
```

**Complessità**

$C[0, k] \leftarrow 0$	$\Theta(k)$
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 CountingSort è la memoria. Infatti, al crescere di  $k$ , la memoria richiesta per allocare  $\mathbf{C}$  cresce esponenzialmente.

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

**4.6.1 Proprietà di Stabilità**

Dato  $A[1..n]$  in input, se  $A[i] \leq A[j]$  con  $i \leq j$ , allora nell'output  $A[i]$  e  $A[j]$  sono nello stesso ordine relativo.

Algoritmi stabili:

- MergeSort
- InsertionSort

Algoritmi non stabili:

- CountingSort
- QuickSort
- HeapSort

## 4.7 Radix Sort

Il **Radix Sort** è un algoritmo di ordinamento in tempo lineare  $O(n)$ , come **CountingSort**, 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**<sup>1</sup>.

(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  $j$  //  $A^j[i] = a_j a_{j-1} \dots a_1$ 
      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.

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

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

Posso distinguere due casi:

---

<sup>1</sup>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.

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

**Costo?**

$$\begin{aligned} d \text{ volte CountingSort } \Theta(n+b) &\Rightarrow \Theta(d(n+b)) = \Theta(n) \\ \text{con } d = \Theta(1), \text{ base } b &= \Theta(n) \end{aligned}$$

## 5 Tabelle Hash

 $U$  universo delle chiavi

$$U = \{0, 1, \dots, |U| - 1\}$$

 $T[0 \dots |U| - 1]$  tabella hash

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

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.

$$\begin{aligned} &2^8 \dots 2^8 \\ (2^8)^8 &= 2^{64} \cong 10^{19} \end{aligned}$$

**Idea**

$$U = \{0, 1, \dots, |U| - 1\}$$
$$T[0 \dots m - 1] \quad m \ll |U|$$

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

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

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.

$n = \# \text{elementi inseriti}$

$m = \text{dimensione di } T$

$\alpha = \frac{n}{m}$  fattore di carico

$\alpha$  può essere  $<$ ,  $=$  oppure  $>$  di 1

### 5.1.1 Hashing uniforme semplice

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

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

Consideriamo  $n_1, n_2, \dots, n_{m-1}$  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.**  $\text{Search}(k)$ ,  $k$  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.**  $\text{Search}(k)$ ,  $k$  presente.
  - $h(k)$  e  $T[h(k)]$

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

Costo della ricerca di  $x_i$ :

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

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

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

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

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

### 5.1.2 Funzioni Hash

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

“Ogni chiave ha la stessa 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

$$U = \{0, 1, \dots, |U| - 1\}$$

$$h(k) = k \mod n$$

- $m = 2^p$  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

$$k \in U$$

$$0 < A < 1 \text{ fissato}$$

$$h(k) = m(kA \mod 1) \quad \text{Miglior } A : \frac{\sqrt{5} - 1}{2}$$

$$m = 2^p \quad w = \# \text{bit parola}$$

$$A = \frac{q}{2^w} \quad 0 < q < 2$$

$$m(kA \mod 1)$$

$$= m \left( k \frac{q}{2^w} \mod 1 \right) \quad (\text{shift di } w \text{ bit, prendo la parte decimale}$$

$$ka \mod 1 \text{ e la moltiplico per } m = 2^p)$$

### 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)**  $\forall k_1, k_2 \in U \ k_1 \neq k_2$

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

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:

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

che è una permutazione di

$$0, 1, \dots, m-1$$

INSERT( $T, x$ )

```

1   $i = 0$ 
2  repeat
3       $j = h(x.key, i)$ 
4      if ( $T[j] = \text{NIL}$ ) or ( $T[j] = \text{DELETED}$ ) // posizione libera
5           $T[j] = x$ 
6          return  $j$ 
7       $i = i + 1$ 
8  until  $i = m$ 
9  error
```

SEARCH( $T, k$ )

```

1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j].key = k$ 
5          return  $j$ 
6       $i = i + 1$ 
7  until ( $i = m$ ) or ( $T[i] = \text{NIL}$ )
8  return NOT FOUND
```

DELETE( $T, j$ )

```

1   $T[j] = \text{DELETED}$ 
```

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  $h'(k)$ .

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

Inserimento di  $k$

$$j = h'(k)$$

$$i = 0$$

**while** ( $i < m$ ) **and** ( $T[j] \neq \text{NIL/DELETED}$ )

$$i = i + 1$$

$$j = (j + 1) \mod n$$

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

$$(i = 1) \quad j = (h'(k) + 1) \mod m$$

$\vdots$

$$(i = l)$$

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



### 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 ( $MCD = 1$ );
- $i, i' < m$   $h(k, i) = h(k, i') \Rightarrow i = i'$  (**iniettività**)

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

**iniettiva  $\Rightarrow$  biiettiva**

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

Scelgo  $m = 2^p$ ,  $h_2(k) = 1 + 2h'_2(k)$ ,  $h'_2(k)$  qualunque.

**es.**  $h_2(k) = 1 + k \mod m'$  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)  $\frac{1}{1-\alpha}$  se  $\alpha < 1$
- (b)  $m$  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  $x_0, x_1, \dots, x_i, \dots, x_n$ .

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

Numero medio:

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

◦ Se  $\alpha < 1$

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

◦ Se  $\alpha = 1$

$$\begin{aligned}
 \sum_{l=1}^m \frac{1}{l} &= 1 + \sum_{l=2}^m \frac{1}{l} \leq \int_1^m \frac{1}{x} dx \\
 &= 1 + (\log m - \log 1) = 1 + \log m
 \end{aligned}$$

Confrontiamo le complessità dei due casi.

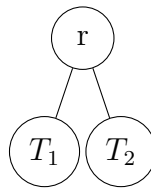
$\alpha$	$\frac{l}{1-\alpha}$	$\frac{1}{\alpha} \log \left( \frac{1}{1-\alpha} \right)$
$\alpha = 0.3$	1.43	1.19
$\alpha = 0.5$	2.00	1.39
$\alpha = 0.7$	3.33	1.72
$\alpha = 0.9$	10	2.56
$\alpha = 0.99$	100	4.65

## 6 Alberi di Ricerca

### 6.1 Alberi Binari di Ricerca (ABR)

**Definizione induttiva**

- $\emptyset$  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:

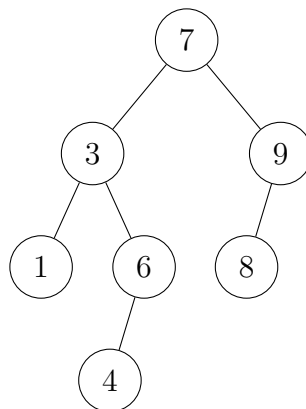
- $x.p$
- $x.key$
- $x.left$
- $x.right$

**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( $x$ )

```

1  if  $x \neq \text{NIL}$ 
2      IN-ORDER( $x.\text{left}$ )
3      PRINT( $x$ ) //  $\Theta(1)$ 
4      IN-ORDER( $x.\text{right}$ )

```

**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).

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

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

( $n > 0$ ) Proprietà vera per  $n' < n$

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

### 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  $x.\text{key} = k \Rightarrow$  restituisce  $x$ ;
- Altrimenti, ricorre sul prossimo nodo.

SEARCH( $x, k$ )

```
1  if ( $x = \text{NIL}$ ) or ( $x.\text{key} = k$ )
2      return  $x$ 
3  else if  $k < x.\text{key}$ 
4      return SEARCH( $x.\text{left}, k$ )
5  else
6      return SEARCH( $x.\text{right}, k$ )
```

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

Vediamo una versione iterativa di **Search**.

SEARCH-IT( $x, k$ )

```
1  while ( $x \neq \text{NIL}$ ) or ( $x.\text{key} \neq k$ )
2      if  $k < x.\text{key}$ 
3           $x = x.\text{left}$ 
4      else
5           $x = x.\text{right}$ 
6  return  $x$ 
```

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

MIN( $T$ )

```
1   $x = T.\text{root}$ 
2  if  $x = \text{NIL}$ 
3      return NIL
4  else
5      while  $x.\text{left} \neq \text{nil}$ 
6           $x = x.\text{left}$ 
7  return  $x$ 
```

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

MAX( $T$ )

```
1   $x = T.\text{root}$ 
2  if  $x = \text{NIL}$ 
3      return NIL
4  else
5      while  $x.\text{right} \neq \text{nil}$ 
6           $x = x.\text{right}$ 
7  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  $\text{MIN}(x)$ ;
- Altrimenti, il successore è l’antenato più vicino di cui  $x$  è nel sottoalbero sinistro.

SUCCESSOR( $x$ )

```
1  if  $x.\text{right} \neq \text{NIL}$ 
2      return MIN( $x.\text{right}$ )
3  else
4       $y = x.p$ 
5      while ( $y \neq \text{NIL}$ ) and ( $x = y.\text{right}$ )
6           $x = y$ 
7           $y = y.p$ 
8      return  $y$ 
```

**Costo?**  $O(h)$

#### 6.1.4 Inserimento

INSERT( $T, z$ )

```
1   $x = T.root$ 
2   $y = NIL$ 
3  while  $x \neq NIL$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else
8           $x = x.right$ 
9   $z.p = y$ 
10 if  $y = NIL$ 
11      $T.root = z$ 
12 else
13     if  $z.key < y.key$ 
14          $y.left = z$ 
15     else
16          $y.right = z$ 
```

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( $T, u, v$ )

```
1  if  $u.p = NIL$ 
2       $T.root = v$ 
3  else
4      if  $u = u.p.left$ 
5           $u.p.left = v$ 
6      else
7           $u.p.right = v$ 
8  if  $v \neq NIL$ 
9       $v.p = u.p$ 
```



```

DELETE( $T, z$ )
1  if  $z.left = \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  else if  $z.right = \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else
6       $y = \text{MIN}(z.right)$ 
7      if  $y.p \neq z$ 
8          TRANSPLANT( $T, y, y.right$ )
9           $y.right = z.right$ 
10          $y.right.p = y$ 
11          $y.left = z.left$ 
12          $y.left.p = y$ 
13         TRANSPLANT( $T, z, y$ )
14     else
15          $y.left = z.left$ 
16          $y.left.p = y$ 
17         TRANSPLANT( $T, z, y$ )

```

È possibile scrivere **Delete** in modo più compatto.

```

DELETE( $T, z$ )
1  if  $z.left = \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  else if  $z.right = \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else
6       $y = \text{MIN}(z.right)$ 
7      if  $y.p \neq z$ 
8          TRANSPLANT( $T, y, y.right$ )
9           $y.right = z.right$ 
10          $y.right.p = y$ 
11          $y.left = z.left$ 
12          $y.left.p = y$ 
13         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  $bh(x)$ ).

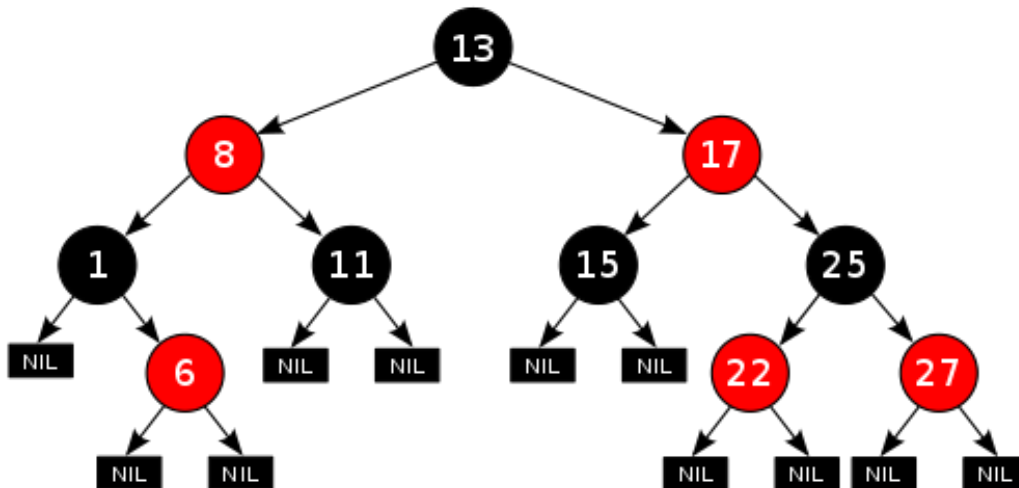


Figura 6: Esempio di un RB-tree.

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 ( $\neq \text{NIL}$ ) 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  $T.nil \Rightarrow n_x = 0 = 2^0 - 1$  ( $2^0$  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:

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

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

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

Complessivamente

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

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

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

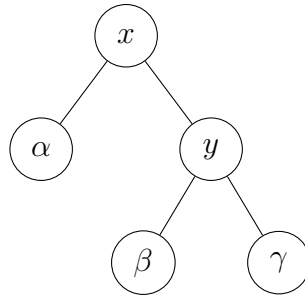
### 6.2.1 Complessità algoritmi RB-Trees

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

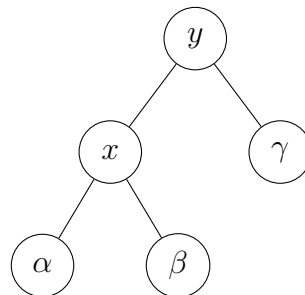
### 6.2.2 RB-Insert e RB-Delete

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  $\alpha$ ,  $\beta$  e  $\gamma$  sono sotto-alberi (il colore dei nodi non ha importanza ai fini della procedura che andremo a vedere)<sup>1</sup>:



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



<sup>1</sup>Di conseguenza, applicandola a un **RB-tree**, gli assiomi di validità potrebbero venire violati.

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

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

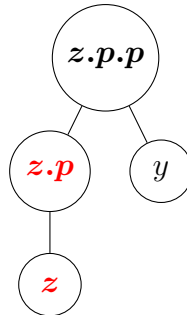
LEFT( $T, x$ )

- 1  $x.right = y.left$
- 2  $x.right.p = x$
- 3  $y.left = x$
- 4  $x.p = y$
- 5 TRANSPLANT( $T, x, y$ )

**RB-Insert( $T, z$ )** Voglio inserire  $z$  nell'albero  $T$ . L'idea è quella di porre  $z.col = \text{RED}$  poichè meno insidioso<sup>1</sup>.

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

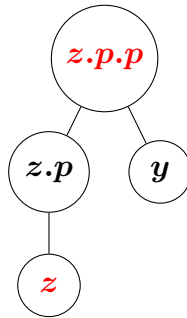
Abbiamo due **macrocasi**.  $z.p$  è figlio sinistro, oppure destro. Noi analizzeremo solo il primo:  **$z.p$  figlio sinistro**.



<sup>1</sup>Andando a modificare il numero di nodi neri, cambia l'altezza nera, e la cosa è difficile da sistemare.

Abbiamo due possibilità per  $y^1$ :

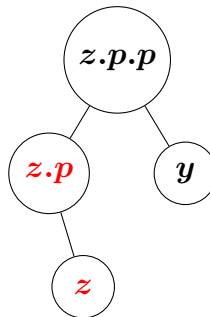
1.  $y.col = \text{RED}$ . Ci è sufficiente invertire il colore di  $z.p.p$  con quello dei figli.



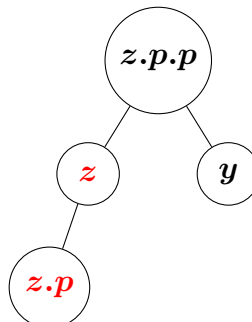
In questo modo, risolviamo localmente e rimandiamo il problema in alto.

2.  $y.col = \text{BLACK}$ . Possiamo distinguere due sottocasi.

(2.1)  $z$  figlio destro.

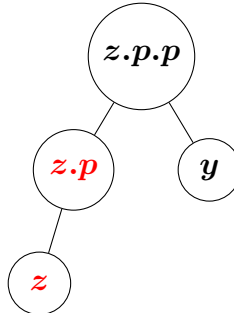


Voglio finire nel caso (2.2). Applico  $\text{Left}(T, z.p)$ , ottenendo:

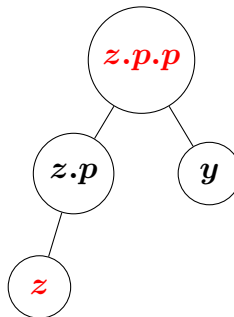


<sup>1</sup>I nodi con testo in rosso sono RED, quelli in grassetto sono BLACK, e quelli normali possono essere sia rossi che neri.

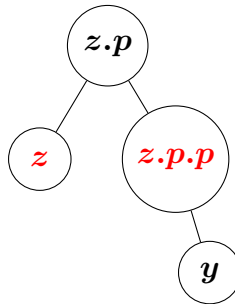
(2.2)  $z$  figlio sinistro.



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



Applico  $\text{Right}(T, z.p.p)$ <sup>1</sup>:



$\text{RB-INSERT}(T, z)$

1  $\text{INSERT}(T, z)$

2  $z.col = \text{RED}$

3  $\text{RB-INSERTFIX}(T, z)$

---

<sup>1</sup>Analoga di **Left**.

RB-FIXUP( $T, z$ )

```

1  while  $z.p.col = \text{RED}$ 
2      if  $z.p = z.p.p.left$  // Macrocaso  $z.p$  figlio sinistro
3           $y = z.p.p.right$ 
4          if  $y.col = \text{RED}$  // Caso 1
5               $z.p.p.col = \text{RED}$ 
6               $z.p.col = \text{BLACK}$ 
7               $y.col = \text{BLACK}$ 
8               $z = z.p.p$ 
9          else // Caso 2
10             if  $z = z.p.right$  // Caso (2.1)
11                 LEFT( $T, z.p$ )
12                  $z = z.left$ 
13             // Caso (2.2)
14              $z.p.col = \text{BLACK}$ 
15              $z.p.p.col = \text{RED}$ 
16             RIGHT( $T, z.p.p$ )
17 else ... // Macrocaso  $z.p$  figlio destro
18  $T.root.col = \text{BLACK}$ 

```

### Costo

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

**RB-Delete( $T, z$ )** La **Delete** è ancora più problematica<sup>1</sup>. 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...).

---

<sup>1</sup>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:

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

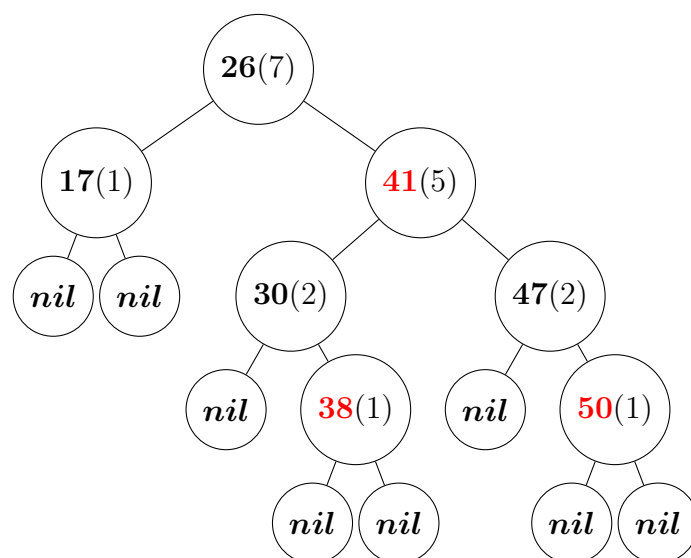
$$x.size = \# \text{nodi radicati nel sottoalbero } T_x$$

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( $x$ )
1  if  $x = T.nil$ 
2       $x.size = 0$ 
3  else
4       $l = \text{SIZE}(x.left)$ 
5       $r = \text{SIZE}(x.right)$ 
6       $x.size = l + r + 1$ 
7  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:  $x \neq T.nil, i : 1 \leq i \leq x.size$ 
1   $r = x.left.size + 1$ 
2  if  $i = r$ 
3      return  $x$ 
4  else if  $i < r$ 
5      return SELECT( $x.left, i$ )
6  else
7      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( $x$ )
1   $r = x.left.size + 1$ 
2   $y = x$ 
3  while  $y.p \neq T.nil$  // idea:  $r$  contiene la posizione di  $x$  in  $T_y$ 
4      if  $y.p.right = y$ 
5           $r = r + y.p.left.size + 1$ 
6       $y = y.p$ 
7  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
    1   $z.size = 1$ 
    2   $x = T.root$ 
    3   $y = T.nil$ 
    4  while  $x \neq T.nil$ 
    5       $x.size = x.size + 1$ 
    6       $y = x$ 
    7      if  $z.key < x.key$ 
    8           $x = x.left$ 
    9      else
    10          $x = x.right$ 
    11   $z.p = y$ 
    12  if  $y = NIL$ 
    13       $T.root = z$ 
    14  else
    15      if  $z.key < y.key$ 
    16           $y.left = z$ 
    17      else
    18           $y.right = z$ 
    // (2) RB-FixUp
    19   $z.col = RED$ 
    20  RB-FIXUP( $T, z$ )

```

E la versione aggiornata di **Left**

```

LEFT( $T, x$ )
    1   $x.right = y.left$ 
    2   $x.right.p = x$ 
    3   $y.left = x$ 
    4   $x.p = y$ 
    5  TRANSPLANT( $T, x, y$ )
    6   $y.size = x.size$ 
    7   $x.size = x.left.size + x.right.size + 1$ 

```

### 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

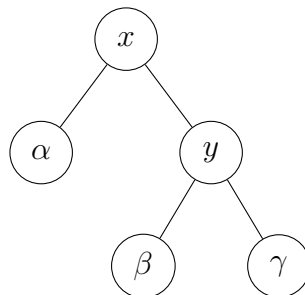
$$x.max = \max \begin{cases} x.left.max \\ x.right.max \\ x.int.high \end{cases}$$

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

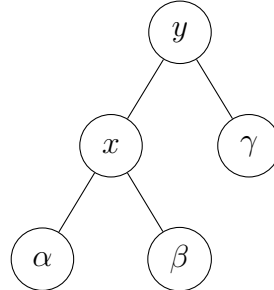
Vogliamo implementare le seguenti procedure:

- **Insert**( $T, x$ )
- **Delete**( $T, x$ )
- **ISearch**( $T, i$ ) con  $i = [low, high]$ :
  - $x$  tale che  $x.int \cap i \neq \emptyset$ ;
  - $T.nil$  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**( $x, i$ )

```

1  if ( $x = T.nil$ ) or ( $x.int \cap i \neq \emptyset$ )
2      return  $x$ 
3  else if ( $x.left$ ) and ( $x.left.max \geq i.low$ )
4      return ISearch( $x.left, i$ )
5  else
6      return ISearch( $x.right, i$ )
  
```

**Correttezza**

- **Else if.** Consideriamo in  $x.left$  un intervallo  $i'$ . Abbiamo 2 possibilità.

- (1)  $i \cap i' \neq \emptyset$
- (2)  $i \cap i' = \emptyset$ , 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.**  $\forall i' \text{ in } x.left \Rightarrow i' \cap i \neq \emptyset$ .

**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”:

$\Downarrow$

$\left\{ \begin{array}{l} \text{top down} \\ \text{bottom up} \end{array} \right.$

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  $p_1, p_2, \dots, p_n$ .
- $r_n$  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?

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

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  $b$ ).

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

**Pseudocodice**

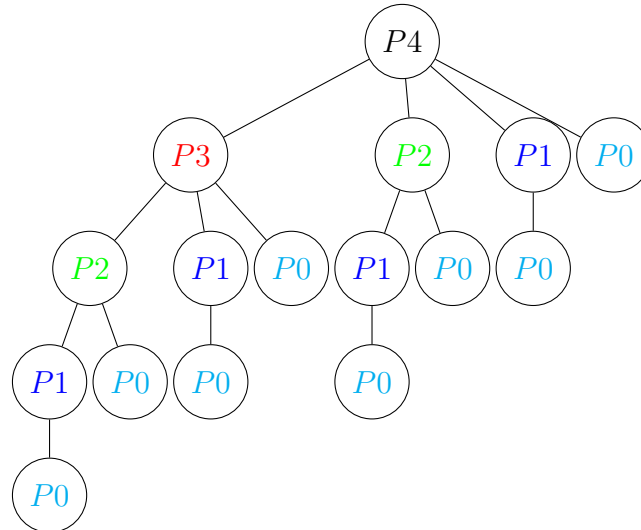
```
CutRod( $p, n$ )
1  if  $n = 0$ 
2      return 0
3  else
4       $q = -1$ 
5      for  $i = 1$  to  $n$ 
6           $q = \text{MAX}(q, P[i] + \text{CutRod}(P, n - i))$ 
7  return  $q$ 
```

**Complessità**

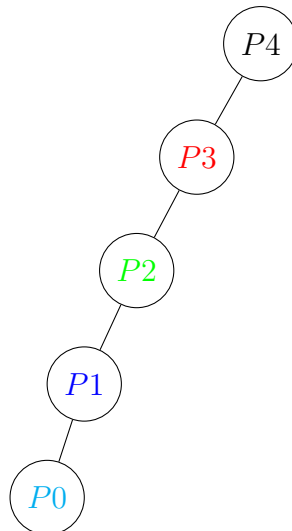
$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^n T(n-i) \\ &= 1 + \sum_{j=0}^{n-1} T(j) = \Theta(2^n) \\ T(0) &= 1 + 0 = 1 = 2^0 \\ T(n+1) &= 1 + \sum_{j=0}^n T(j) \\ &= 1 + \sum_{j=0}^{n-1} T(n) + T(n) \quad \left( \sum_{j=0}^{n-1} T(n) = T(n) \right) \\ &= 2T(n) \end{aligned}$$



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( $p, n$ )

```

1  for  $i = 1$  to  $n$ 
2       $r[i] = -1$ 
3  return MEMCUTROD( $p, n, r$ )

```

MEMCUTROD-AUX( $p, j, r$ )

```

1  if  $r[j] < 0$ 
2      if  $j = 0$ 
3           $r[j] = 0$ 
4      else
5           $q = -1$ 
6          for  $i = 1$  to  $j$ 
7               $q = \text{MAX}(q, p[i] + \text{MEMCUTROD-AUX}(p, j - i, r))$ 
8           $r[j] = q$ 
9  return  $r[j]$ 

```

$$j = 1, \dots, n$$

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

**Bottom-Up**BOTTOMUPCUTROD( $p, n$ )

```

1  alloco  $r[1..n]$ 
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$  // calcolo  $r[j]$ 
4       $q = -1$ 
5      for  $i = 1$  to  $j$ 
6           $q = \text{MAX}(q, p[i] + r[j - 1])$  //  $\Theta(1)$   $r[j] = q$ 
7  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  $p \times r$ .

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

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

Vogliamo moltiplicare  $n$  matrici  $C = A_1 \times A_2 \times \dots \times A_n$ . 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**:

$$(A_1 \times A_2) \times A_3 \quad (10 \times 100 \times 5) + 100 \times 5 \times 50 = 7500$$

$$A_1 \times (A_2 \times A_3) \quad 10 \times 100 \times 50 + (100 \times 5 \times 50) = 75000$$

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

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

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**  $A_i \times \dots \times A_j$ ,  $1 \leq i < j \leq n$ , in numero  $\Theta(n^2)$

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

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

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  $A_{ij} = (A_1 \times A_2 \times \dots \times A_j)$

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

$m[i, j]$  = minimo # di prodotti per calcolare  $A_{ij}$

$$= \begin{cases} 0 & i = j \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j) & i < j \end{cases}$$

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

### Pseudocodice

MATRIXCHAIN( $p, i, j$ )

```

1  if  $i = j$ 
2      return 0
3  else
4       $cmin = +\infty$ 
5      for  $k = i$  to  $j - 1$ 
6           $q = \text{MATRIXCHAIN}(p, i, k) + \text{MATRIXCHAIN}(p, k + 1, j) +$ 
               $+ p[i - 1]p[k]p[j]$ 
7          if  $q < cmin$ 
8               $cmin = q$ 
9  return  $cmin$ 
```

**Complessità** Mi aspetto poca efficienza.

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

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

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

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

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( $p, i, j$ )

```

1  for  $i = 1$  to  $n$ 
2      for  $j = 1$  to  $n$ 
3           $m[i, j] = +\infty$ 
4  return MATRIXCHAINREC( $p, 1, n, m$ )
```

MATRIXCHAINREC( $p, i, j, m$ )

```

1  if  $m[i, j] = +\infty$ 
2      if  $i = j$ 
3           $m[i, j] = 0$ 
4      else
5          for  $k = 1$  to  $j - 1$ 
6               $q = \text{MATRIXCHAINREC}(p, i, k, m) +$ 
                   $+\text{MATRIXCHAINREC}(p, k + 1, j, m) +$ 
                   $+p[i - 1]p[k]p[j]$ 
7              if  $q < m[i, j]$ 
8                   $m[i, j] = q$ 
9  return  $m[i, j]$ 
```

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

$\Rightarrow$  complessità  $O(n^3)$

◦ **Soluzione Bottom-up**

per lunghezza crescente della sequenza  $A_i \dots A_j$

MATRIXCHAIN( $p, n$ )

```

1  for  $i = 1$  to  $n$ 
2       $m[i, j] = 0$ 
3  for  $l = 2$  to  $n$ 
4      for  $i = 1$  to  $n - l + 1$ 
5           $j = i + l - 1$ 
6           $m[i, j] = +\infty$ 
7          for  $k = 1$  to  $j - 1$ 
8               $q = m[i, j] + m[k + 1, j] + p[i - 1]p[k]p[j]$ 
9              if  $q < m[i, j]$ 
10                  $m[i, j] = q$ 
11 return  $m[1, n]$ 
```

Complessità  $O(n^3)$

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

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

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

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

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

**Stampa del risultato**

$S[i, j]$  = posizione della parentetizzazione più esterna per  $A_i \dots A_j$

PRINTPAREN( $S, i, j$ )

```
1  if  $i = j$ 
2      PRINT " $A_i$ "
3  else
4      PRINT "("
5      PRINTPAREN( $S, i, S[i, j]$ )
6      PRINTPAREN( $S, S[i, j] + 1, j$ )
7      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

$$\begin{aligned}x_1 : & \quad A C T A C C T G \\x_2 : & \quad A T C A C C\end{aligned}$$

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  $x_{i_1} \dots x_{i_k}$  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  $X^k = x_1 \dots x_k$   $k \leq m$  (ad esempio  $X^0$  è  $\varepsilon$ )



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.

$$X = x_1 \dots x_{m-1} x_m$$

$$Y = y_1 \dots y_{n-1} y_n$$

$$\circ w_k = x_m = y_n$$

$$\text{Se } w_k \neq x_m \Rightarrow W \text{ sottoseq. di } X^{m-1}, Y^{n-1}$$

Impossibile,  $Wx_m$  sarebbe sottoseq. di  $X$  e  $Y$  più lunga di  $W$

$$\circ W^{k-1} \in LCS(X^{m-1}, Y^{n-1})$$

$W^{k-1}$  se non ottima, esisterebbe  $W'$  sottoseq. di  $X^{m-1}$  e  $Y^{n-1}$  tale che  $|W'| > |W^{k-1}|$ , ottenendo  $W'x_m$  sottoseq. di  $X$  e  $Y$  con  $|W'x_m| > |W^{k-1}x_m| = |W|$ , **assurdo**.

2.  $x_m \neq y_n$

(2a)  $x_m \neq w_k$

$$\Rightarrow W \text{ è sottoseq. di } X^{m-1}$$

$$W \text{ è sottoseq. di } Y$$

$$W \in LCS(X^{m-1}, Y)? \quad (\text{è ottima?})$$

Se no, 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( $X, Y$ )

```

1   $m = X.\text{len}, n = Y.\text{len}$ 
2  for  $i = 0$  to  $m$  //  $\Theta(m)$ 
3       $C[i, 0] = 0$ 
4  for  $j = 1$  to  $n$  //  $\Theta(n)$ 
5       $C[0, j] = 0$ 
6  for  $i = 1$  to  $m$  //  $\Theta(n \cdot m)$ 
7      for  $j = 1$  to  $n$ 
8          if  $X[i] = Y[j]$  // C'è un match
9               $C[i, j] = C[i-1, j-1]$ 
10              $b[i, j] = \nwarrow$ 
11         else
12             if  $C[i-1, j] \geq C[i, j-1]$ 
13                  $C[i, j] = C[i-1, j]$ 
14                  $b[i, j] = \leftarrow$ 
15             else
16                  $C[i, j] = C[i, j-1]$ 
17                  $b[i, j] = \uparrow$ 
18  return  $b, c$ 
```

PRINTLCS( $X, Y$ )

```

1   $b, C = \text{LCS}(X, Y)$ 
2  PRINTLCSREC( $X, b, m, n$ )
```

```
PRINTLCSREC( $X, b, i, j$ )
1  if  $i > 0$  and  $j > 0$ 
2      if  $b[i, j] = \nwarrow$ 
3          PRINTLCSREC( $X, b, i - 1, j - 1$ )
4          PRINT $X[i]$ 
5      else if  $b[i, j] = \leftarrow$ 
6          PRINTLCSREC( $X, b, i - 1, j$ )
7      else
8          PRINTLCSREC( $X, b, i, j - 1$ )
```

# Appendices

## A Raccolta algoritmi

### A.1 Insertion Sort

Per approfondire, vedi la sezione 2.2

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       $A[i+1] = key$ 
```

### A.2 Merge Sort

Vedi la sezione 2.4

MERGESORT( $A, p, r$ )

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

```

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

```

### A.3 Insertion Sort ricorsivo

```

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

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

```

#### A.3.1 Correttezza di 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. Dopodiché, chiamo **Insert**( $A, j-1$ ) per ordinare la cella  $A[j-1]$ .

**A.4 CheckDup**

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

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

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

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

```

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

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

```

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

```

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

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

## A.5 SumKey

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

→ incremento *i*;

(1) **non** cambia;

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

Distinguiamo 2 casi.



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

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

- $k \in [j + 1, n]$  quindi

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

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

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

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

$$A[i] \geq 0 \quad \forall i \in [1, n], \quad key \leq top$$

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

SUMV(*A*, *key*)

```

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

Complessità:

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

## A.6 Heap Sort

Per approfondire, vedi 4.1.

LEFT( $i$ )

// restituisce il figlio sx del nodo  $i$

1 **return**  $2 * i$

RIGHT( $i$ )

// restituisce il figlio dx del nodo  $i$

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

PARENT( $i$ )

// restituisce il genitore del nodo  $i$

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

MAXHEAPIFY( $A, i$ )

1  $l = \text{LEFT}(i)$

2  $r = \text{RIGHT}(i)$

3 **if** ( $l \leq A.\text{heapsize}$ ) **and** ( $A[l] > A[i]$ )

4      $max = l$

5 **else**

6      $max = i$

7 **if** ( $r \leq A.\text{heapsize}$ ) **and** ( $A[r] > A[max]$ )

8      $max = r$

9 **if** ( $max \neq i$ )

10      $A[i] \leftrightarrow A[max]$

11     MAXHEAPIFY( $A, max$ )

BUILDMAXHEAP( $A$ )

1  $A.\text{heapsize} = A.\text{length}$

2 **for**  $i = \lfloor A.\text{length}/2 \rfloor$  **down to** 1

3     MAXHEAPIFY( $A, i$ )

HEAPSORT( $A$ )

1 BUILDMAXHEAP( $A$ ) //  $O(n)$

2 **for**  $i = A.\text{length}$  **down to** 2

3      $A[1] \leftrightarrow A[i]$

4      $A.\text{heapsize} = A.\text{heapsize} - 1$

5     MAXHEAPIFY( $A, 1$ ) //  $O(\log n)$

## A.7 Code con Priorità

(Sezione 4.1.2)

MAX(*A*)

```
1  if A.heapsize = 0
2      error
3  else return A[1]
```

EXTRACTMAX(*A*)

```
1  max = A[1]
2  A[1] = A[A.heapsize]
3  A.heapsize = A.heapsize - 1
4  MAXHEAPIFY(A, 1) // ripristina le proprietà di MaxHeap
5  return max
```

MAXHEAPIFYUP(*A*, *i*)

```
1  if (i > 1) and (A[i] > A[PARENT(i)])
2      A[i] ↔ A[PARENT(i)]
3      MAXHEAPIFYUP(A, PARENT(i))
```

INSERT(*A*, *x*)

```
1  A.heapsize = A.heapsize + 1
2  A[A.heapsize] = x
3  MAXHEAPIFYUP(A, A.heapsize)
```

INCREASEKEY(*A*, *i*,  $\delta$ )

```
    // Precondizione:  $\delta \geq 0$ 
1  A[i] = A[i] +  $\delta$ 
2  MAXHEAPIFYUP(A, i)
```

CHANGEKEY(*A*, *i*,  $\delta$ )

```
1  A[i] = A[i] +  $\delta$ 
2  if  $\delta > 0$ 
3      MAXHEAPIFYUP(A, i)
4  else //  $\delta \leq 0$ 
5      MAXHEAPIFY(A, i)
```

DELETEKEY( $A, i$ )

```
1   $old = A[i]$ 
2   $A[i] = A[A.heapsize]$ 
3   $A.heapsize = A.heapsize - 1$ 
4  if  $old \leq A[i]$ 
5      MAXHEAPIFYUP( $A, i$ )
6  else
7      MAXHEAPIFY( $A, i$ )
```

## B Esercizi

### B.1 Ricorrenze

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

Esplicitando il caso base della ricorrenza otteniamo:

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

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

$$\begin{aligned} (a = 1) \quad T(n) &= nb + c = \Theta(n) \\ (a < 1) \quad T(n) &= \frac{1-a^n}{1-a} \cdot b + a^n c = \Theta(1) \\ & \quad (\text{valgono } \frac{1-a^n}{1-a} \leq \frac{1}{1-a}, \quad a^n c < c) \\ (a > 1) \quad T(n) &= \frac{a^n-1}{a-1}b + a^n c = \Theta(a^n) \end{aligned}$$

### 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

$$\begin{aligned}
 A[n-1] - A[1] &= (A[n] - A[1]) - (A[n] - A[n-1]) \\
 (\text{valgono } A[n-1] - A[1] &\geq n \text{ e } A[n] - A[n-1] \leq 1) \\
 A[n-1] - A[1] &\geq n-1 \Rightarrow A[1..n-1] \text{ contiene un } \mathbf{gap}
 \end{aligned}$$

```

GAP(A, p, r)
  // pre: r - p > 1
1  if p = r + 1
2      return p
3  else
4      q = (p+r)/2
5      if A[q] - A[p] > q - p + 1
6          GAP(A, p, r)
7      else
8          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)
1  BUILDMAXHEAP(A) // Θ(n)
2  for i = 1 to k
3      x = EXTRACTMIN(A) // k · O(log n)
4  return x

```

Complessità:  $O(n + k \log n)$

(2) Soluzione alternativa.

```

SELECT( $A, k$ )
1  BUILDMAXHEAP( $A, k$ ) //  $\Theta(k)$ 
2  for  $i = k + 1$  to  $n$ 
3      if  $A[i] < A[1]$  //  $O((n - k) \log k)$ 
4           $A[i] \leftrightarrow A[1]$ 
5          MAXHEAPIFY( $A, 1, k$ )
6  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  $A$

```

1  if  $p < r$ 
2       $q = \frac{p+r}{2}$ 
3      return INV( $A, p, q$ ) + INV( $A, q + 1, r$ ) + MERGEINV( $A, p, q, r$ )
4  else
5      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( $A, p, q, r$ )

```

1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3   $L[1, n_1] = A[p, q]$ 
4   $R[1, n_2] = A[q + 1, r]$ 
5   $L[n_1 + 1] = R[n_2 + 1] = \infty$ 
6   $i = j = 1$ 
7   $inv = 0$ 
8  for  $k = p$  to  $r$ 
9      if  $L[i] \leq R[j]$ 
10          $A[k] = L[i]$ 
11          $i = i + 1$ 
12     else
13          $A[k] = R[j]$ 
14          $j = j + 1$ 
15          $inv = inv + n_1 - i + 1$ 
16 return  $inv$ 
```

### B.3 Esercizio del 03/05/2018

ABR con:

- $x.left$
- $x.right$
- $x.succ$
- ~~$x.p$~~ , non ho il padre.

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



INSERT( $T, z$ )

```

1   $x = T.root$ 
2   $y = \text{NIL}$  // parent
3   $pred = \text{NIL}$ 
4  while  $x \neq \text{NIL}$ 
5       $y = x$ 
6      if  $z.key < x.key$ 
7           $x = x.left$ 
8      else
9           $x = x.right$ 
10          $pred = y$ 
11 if  $y = \text{NIL}$ 
12      $T.root = z$ 
13 else if  $z.key < y.key$ 
14      $y.left = z$ 
15 else
16      $y.right = z$ 
17 if  $pred \neq \text{NIL}$ 
18      $z.succ = pred.succ$ 
19      $pred.succ = z$ 
20 else
21      $z.succ = y$ 
```

**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  $y$ .

$$z.succ = y.succ \quad y.succ = z$$

b)  $z$  figlio sinistro di  $y$ .

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

**Costo di Insert**  $O(h)$

SUCC( $T, z$ )

```

1  return  $x.succ$ 
```

```
PARENT( $T, x$ )
1   $y = x$ 
2  while  $y.right \neq \text{NIL}$ 
3       $y = y.right$ 
4   $z = z.succ$ 
5  if  $z = \text{NIL}$ 
6       $w = T.root$ 
7  else
8       $w = z.left$ 
9  if  $w = x$ 
10     return  $z$ 
11 while  $w.right \neq x$ 
12      $w = w.right$ 
13 return  $w$ 
```

**Costo di Parent**  $O(h)$ .

```
PRED( $T, z$ )
1  if  $x.left \neq \text{NIL}$ 
2      return MAX( $x.left$ )
3  else
4      PARENT( $T, x$ )
5      while ( $p \neq \text{NIL}$ ) and ( $x = p.left$ )
6           $x = p$ 
7           $p = \text{PARENT}(T, x)$ 
8      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  $O(h^2)$ .