

UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

# Appunti di Algoritmi e Strutture Dati

a.a. 2018/2019

Autori:

**Timoty Granziero**  
**Marco Siragna**

Repository:

<https://github.com/Marco305/ASD-Notes>

12 giugno 2019

# Indice

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduzione</b>  | <b>1</b>  |
| 1.1      | Problem Solving . . . . .  | 1         |
| 1.2      | Analisi . . . . .  | 2         |
| <b>2</b> | <b>Ordinamento</b>   | <b>3</b>  |
| 2.1      | Problema dell'Ordinamento (Sorting) . . . . .  | 3         |
| 2.2      | Insertion Sort . . . . .   | 3         |
| 2.2.1    | Correttezza di Insertion Sort . . . . .  | 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 . . . . .  | 14        |
| <b>3</b> | <b>Complessità Asintotica</b>  | <b>15</b> |
| 3.1      | Notazione Asintotica . . . . .   | 15        |
| 3.1.1    | Limite Asintotico Superiore . . . . .  | 16        |
| 3.1.2    | Limite Asintotico Inferiore . . . . .  | 18        |
| 3.1.3    | Limite Asintotico Stretto . . . . .  | 19        |
| 3.2      | Metodo del Limite . . . . .  | 20        |
| 3.3      | Proprietà Generali . . . . .   | 21        |
| 3.4      | Complessità di un Problema . . . . .   | 22        |
| 3.4.1    | Esempio: limite inferiore per ordinamento basato su<br>scambi di elementi contigui . . . . . | 22        |
| 3.5      | Soluzione di Ricorrenze . . . . .  | 23        |
| 3.5.1    | Metodo di Sostituzione . . . . .   | 23        |
| 3.5.2    | Master Theorem . . . . .   | 29        |
| <b>4</b> | <b>Ordinamento (cont.)</b>   | <b>34</b> |
| 4.1      | Heapsort . . . . .   | 34        |
| 4.1.1    | Max Heap . . . . .   | 36        |
| 4.1.2    | Code con Priorità . . . . .  | 39        |
| 4.2      | Quicksort . . . . .  | 42        |
| 4.2.1    | Correttezza di Quicksort . . . . .   | 43        |
| 4.2.2    | Complessità di Quicksort . . . . .   | 44        |
| 4.3      | Quicksort a Tre Partizioni . . . . .   | 46        |
| 4.4      | Limite Inferiore . . . . .   | 47        |
| 4.5      | Albero di Decisione . . . . .  | 47        |

|          |  |           |
|----------|--|-----------|
| 4.6      | Counting Sort . . . . .                        | 48        |
| 4.6.1    | Proprietà di Stabilità . . . . .               | 50        |
| 4.7      | Radix Sort . . . . .                           | 51        |
| <b>5</b> | <b>Tabelle Hash</b>                            | <b>53</b> |
| 5.1      | Chaining . . . . .                             | 54        |
| 5.1.1    | Hashing Uniforme Semplice . . . . .            | 54        |
| 5.1.2    | Funzioni Hash . . . . .                        | 56        |
| 5.1.3    | Hashing Universale . . . . .                   | 57        |
| 5.2      | Open Addressing . . . . .                      | 58        |
| 5.2.1    | Hashing Uniforme . . . . .                     | 58        |
| 5.2.2    | Funzioni di Hash . . . . .                     | 59        |
| <b>6</b> | <b>Alberi</b>                                  | <b>62</b> |
| 6.1      | Alberi Binari di Ricerca . . . . .             | 62        |
| 6.1.1    | Visita Simmetrica . . . . .                    | 63        |
| 6.1.2    | Ricerca . . . . .                              | 63        |
| 6.1.3    | Successore . . . . .                           | 65        |
| 6.1.4    | Inserimento . . . . .                          | 66        |
| 6.1.5    | Eliminazione . . . . .                         | 66        |
| 6.2      | Red-Black Trees . . . . .                      | 68        |
| 6.2.1    | Complessità Algoritmi RB-Trees . . . . .       | 70        |
| 6.2.2    | RB-Insert e RB-Delete . . . . .                | 70        |
| 6.3      | Arricchimento di Strutture Dati . . . . .      | 80        |
| 6.3.1    | Statistiche d'Ordine . . . . .                 | 80        |
| 6.3.2    | Teorema dell'Aumento degli RB-Trees . . . . .  | 84        |
| 6.3.3    | Interval Trees . . . . .                       | 84        |
| <b>7</b> | <b>Programmazione Dinamica</b>                 | <b>87</b> |
| 7.1      | Critica al Divide & Conquer (D&C) . . . . .    | 87        |
| 7.2      | Memoizzazione . . . . .                        | 89        |
| 7.3      | Problemi di Ottimizzazione . . . . .           | 91        |
| 7.4      | Problemi su Stringhe . . . . .                 | 92        |
| 7.5      | Longest Common Subsequence (LCS) . . . . .     | 94        |
| 7.5.1    | Proprietà di Sottostruttura Ottima . . . . .   | 95        |
| 7.5.2    | Ricorrenza sui Costi . . . . .                 | 98        |
| 7.6      | Longest Increasing Subsequence (LIS) . . . . . | 102       |
| 7.6.1    | Proprietà di Sottostruttura Ottima . . . . .   | 103       |
| 7.6.2    | Ricorrenza sui Costi . . . . .                 | 104       |
| 7.7      | Completamento a Palindromo (CP) . . . . .      | 106       |
| 7.7.1    | Proprietà di Sottostruttura Ottima . . . . .   | 106       |

---

|       |                                      |     |
|-------|--------------------------------------|-----|
| 7.7.2 | Ricorrenza sulle Lunghezze . . . . . | 107 |
|-------|--------------------------------------|-----|

# 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 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.

INSERTION-SORT( $A$ )

```

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

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

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

### 2.2.1 Correttezza di Insertion Sort

**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:

INSERTION-SORT( $A$ )

```

1   $n = A.length$ 
2  for  $j = 2$  to  $n$  // il primo elemento è già ordinato
3       $key = A[j]$  //  $A[1 \dots j-1]$  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 della procedura, `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**.

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

```

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

```

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

```

1   $n_1 = q - p + 1$  // gli indici partono da 1
2   $n_2 = r - q$ 
   // L sotto-array sx, R sotto-array dx
3  for  $i = 1$  to  $n_1$ 
4       $L[i] = A[p + i - 1]$ 
5  for  $j = 1$  to  $n_2$ 
6       $R[j] = A[q + j]$ 
7   $L[n_1 + 1] = R[n_2 + 1] = \infty$ 
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  $R[1..j-1]$ .

$$\begin{aligned}
 A[p..k-1] &\leq L[i..n_1], R[j..n_2] \\
 &\Downarrow \\
 A[p..k-1] &= A[p..r+1-1] \implies A[p..r] \text{ ordinato}
 \end{aligned}$$

**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

---

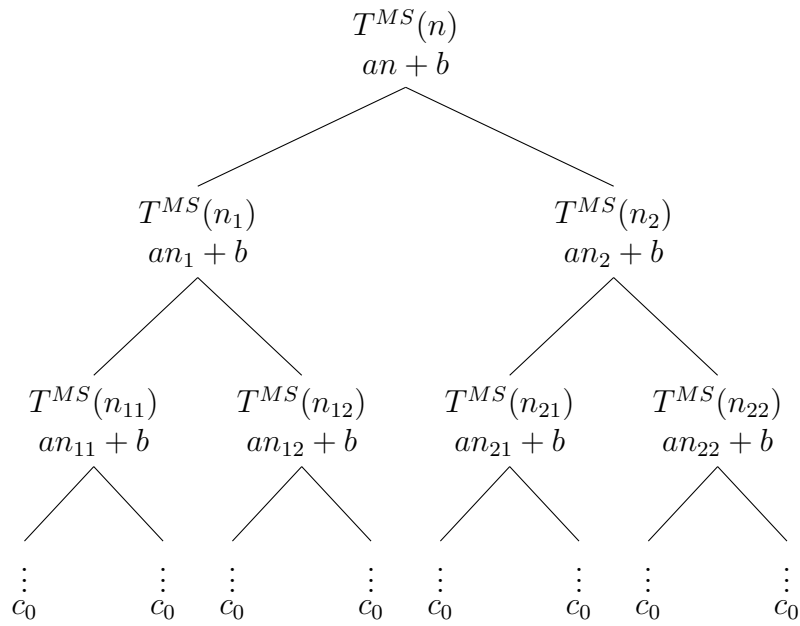
<sup>1</sup>Il simbolo  $\#$  verrà usato per indicare la cardinalità di un insieme.



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

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

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



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.

$$i = 0 \quad an + b$$

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

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

...

$$i = h \quad c_0 n$$

Poniamo  $n = 2^h$ . Abbiamo

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

## 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                   |
| $10^6$ | 17 minuti         | 20ms                   |
| $10^9$ | 70 anni           | 30s                    |

## 3 Complessità Asintotica

### 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 **InsertionSort** e **MergeSort**:

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

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:

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

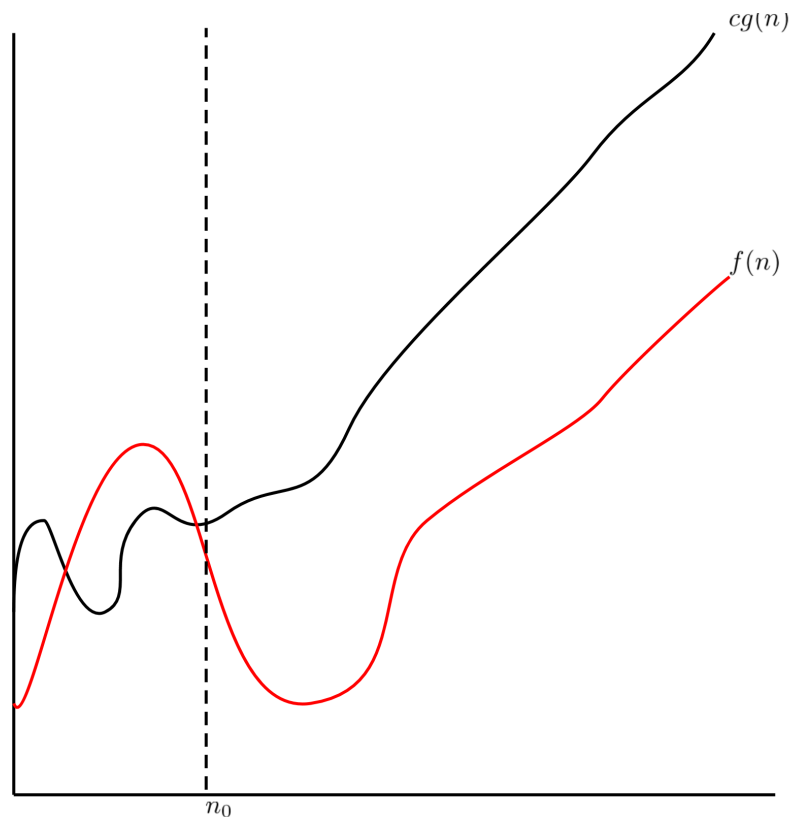


Figura 2: Rappresentazione del limite asintotico superiore per  $f(n)$

#### Esempi

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

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

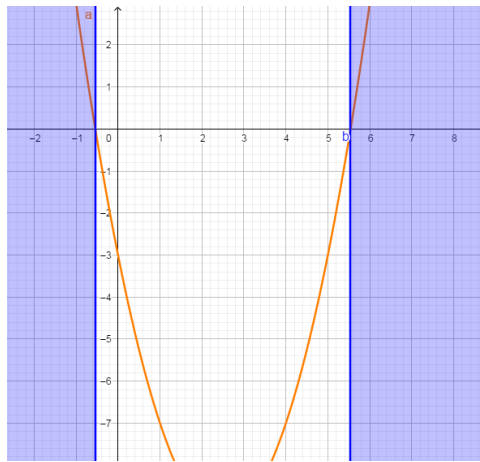
Ipotizziamo  $c = 3$

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

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

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

(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:

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

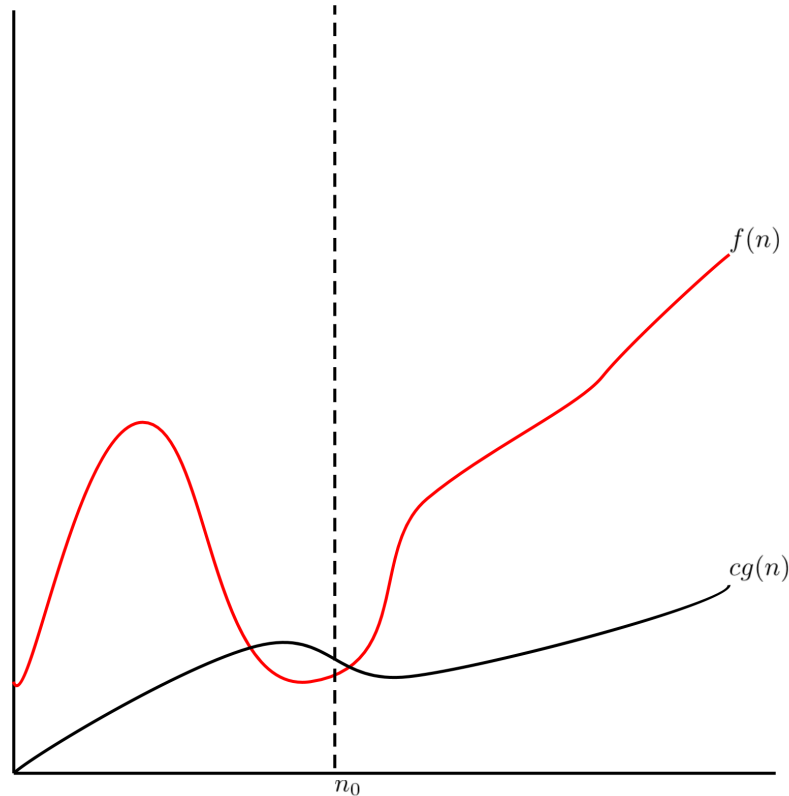


Figura 3: Rappresentazione del limite asintotico inferiore per  $f(n)$

#### 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:

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

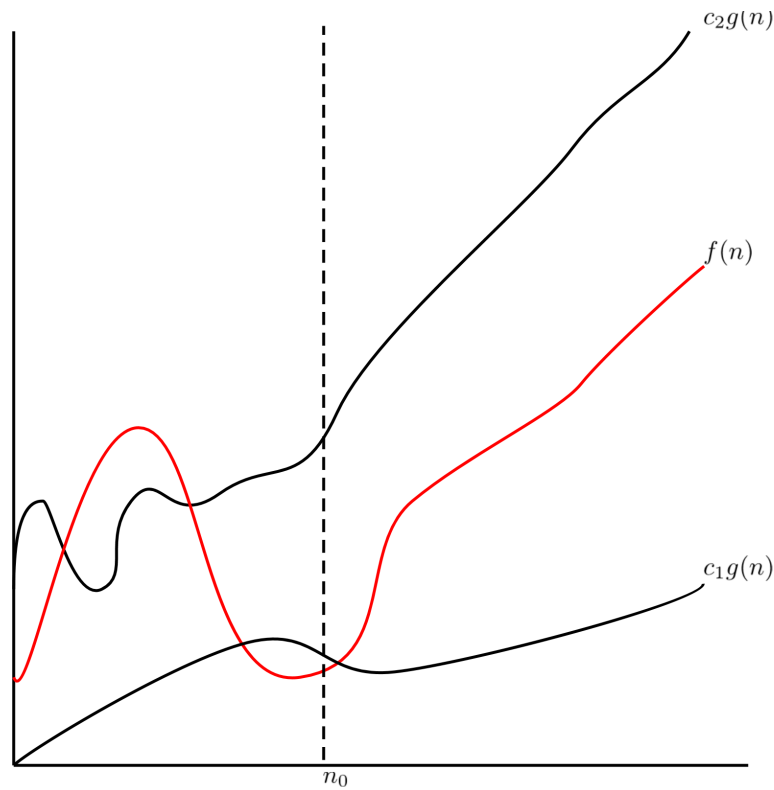


Figura 4: Rappresentazione del limite asintotico stretto per  $f(n)$

#### Esempi

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

$$\begin{aligned} f_1(n) &\neq \Theta(n^3) \\ f_1(n) &= O(n^3) \\ f_1(n) &\neq \Omega(n^3) \\ &\Downarrow \\ \frac{f_1(n)}{n^3} &\rightarrow 0 \end{aligned}$$

### 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**

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

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.

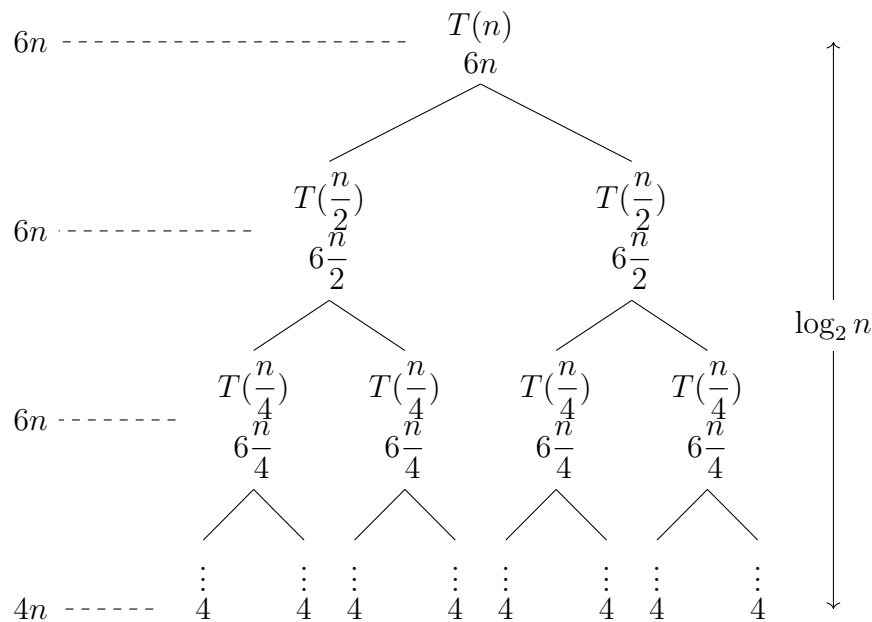
**Esempio**

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

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 **MergeSort**, 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

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

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

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

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).

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

$$2. \quad T(n) = \Omega(n \log n) \text{ è analoga.}$$

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

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

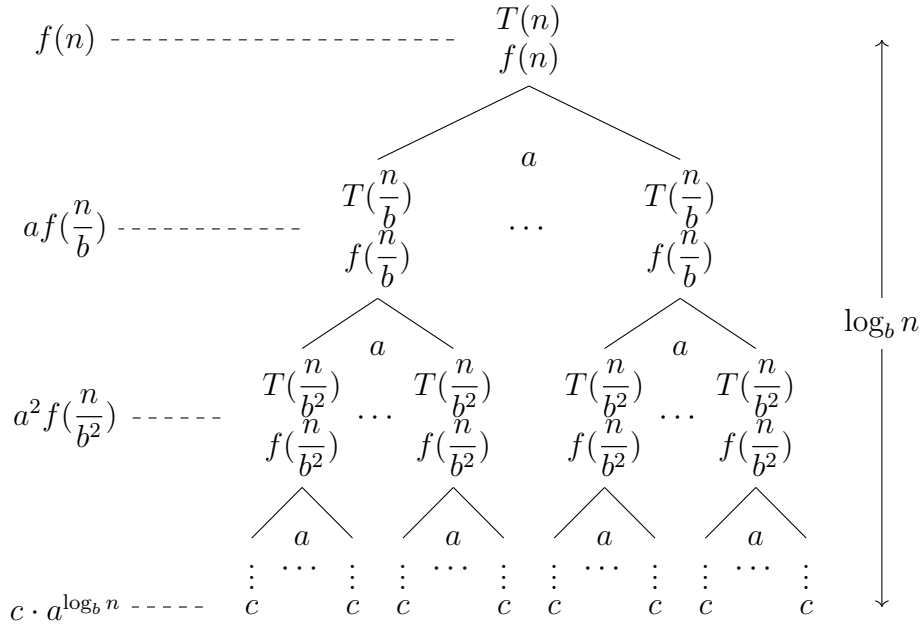
$$T(n) = \Theta(f(n))$$

**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)

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

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

B)

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

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

C)

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

$$\& \mathbf{Regolarità} \Rightarrow \mathbf{Caso\ 3:} \quad T(n) = \Theta(f(n))$$

**Esercizi**

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

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

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

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

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 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 Heapsort

L'**Heapsort**<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 L<sup>A</sup>T<sub>E</sub>X 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 ⌊i/2⌋
```

### 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  $i$ ).

MAX-HEAPIFY( $A, i$ )

```

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     MAX-HEAPIFY( $A, max$ )
```



**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  $i \leq \lfloor \frac{n}{2} \rfloor$

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

**BUILD-MAX-HEAP( $A$ )**

```

1   $A.heapsize = A.length$ 
2  for  $i = \lfloor A.length/2 \rfloor$  down to 1
3      MAX-HEAPIFY( $A, i$ )

```

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  BUILD-MAX-HEAP( $A$ ) //  $O(n)$ 
2  for  $i = A.length$  down to 2
3       $A[1] \leftrightarrow A[i]$ 
4       $A.heapsize = A.heapsize - 1$ 
5      MAX-HEAPIFY( $A, 1$ ) //  $O(\log n)$ 
```

**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  $\Theta(1)$ .

`EXTRACT-MAX( $A$ )`

```
1   $max = A[1]$ 
2   $A[1] = A[A.heapsize]$ 
3   $A.heapsize = A.heapsize - 1$ 
4  MAX-HEAPIFY( $A, 1$ ) // 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`.

MAX-HEAPIFY-UP( $A, i$ )

```

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

```

### 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  MAX-HEAPIFY-UP( $A, A.\text{heapsize}$ )

```

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

INCREASE-KEY( $A, i, \delta$ )

// **Precondizione:**  $\delta \geq 0$

```

1   $A[i] = A[i] + \delta$ 
2  MAX-HEAPIFY-UP( $A, i$ )

```

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

CHANGE-KEY( $A, i, \delta$ )

```

1   $A[i] = A[i] + \delta$ 
2  if  $\delta > 0$ 
3      MAX-HEAPIFY-UP( $A, i$ )
4  else //  $\delta \leq 0$ 
5      MAX-HEAPIFY( $A, i$ )

```

**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**

**DELETE-KEY**( $A, i$ )

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

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

## 4.2 Quicksort

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

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

Si basa sul paradigma del *divide et impera*:

- *Divide*
  - 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

**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 Quicksort

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.

**RANDOMIZED-PARTITION**( $A, p, r$ )

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

### 4.3 Quicksort 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       $q_1, q_2 = \text{TRIPARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q_1 - 1$ )
4      QUICKSORT( $A, q_2 + 1, r$ )
```

## 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)

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

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

### 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.

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

$$h \geq \log_2 n!$$

$$\geq \log_2 \left( n(n-1)(n-2) \dots \frac{n}{2} \right)$$

$$\geq \log_2 \left( \frac{n}{2} \left( \frac{n}{2} - 1 \right) \left( \frac{n}{2} - 2 \right) \dots \frac{n}{2} \right)$$

$$\geq \log_2 \left( \frac{n}{2} \right)^{\left( \frac{n}{2} \right)} = \frac{n}{2} (\log_2 n - \log_2 2) = \frac{n}{2} (\log_2 n - 1) = \Theta(n \log n)$$

Inoltre,

- $\# \text{ operazioni} \geq h = \Omega(n \log n)$
- Heapsort, MergeSort  $O(n \log n)$

$\Rightarrow$  ordinamento (basato 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  $\Omega(n)$

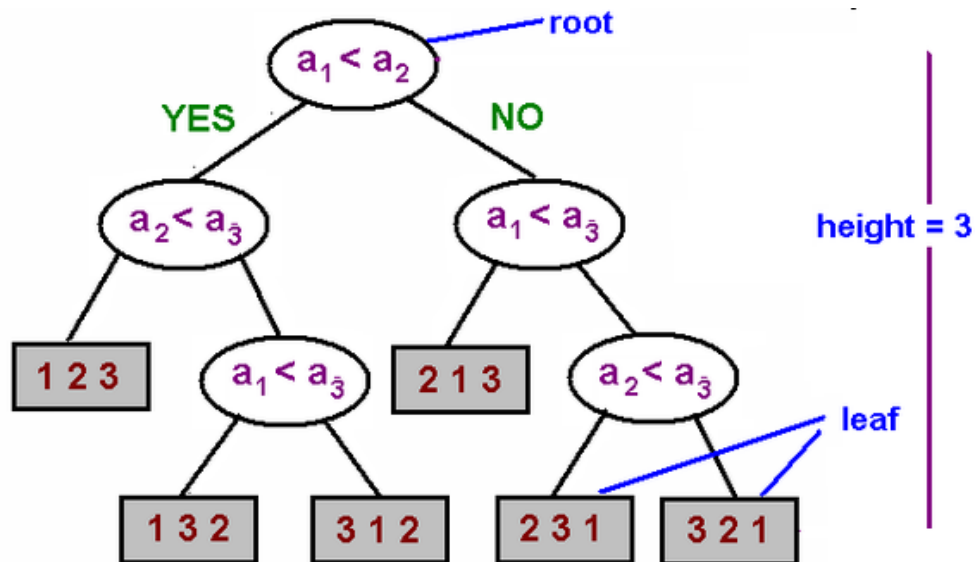


Figura 5: Albero di decisione per l'array  $A[1, 2, 3]$

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

COUNTING-SORT( $A, B, k$ )

```

1   $C[0..k] \leftarrow 0$ 
2  for  $j = 1$  to  $A.length$ 
    //  $C[x] = \#$  elementi in  $A$  con valore  $x$ 
3     $C[A[j]] = C[A[j]] + 1$ 
4  for  $i = 1$  to  $k$ 
    //  $C[x] = \#$  elementi 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**

|                               |             |
|-------------------------------|-------------|
| $C[0, k] \leftarrow 0$        | $\Theta(k)$ |
| <b>for</b> $j=1 \dots$        | $\Theta(n)$ |
| <b>for</b> $i=1 \dots$        | $\Theta(k)$ |
| <b>for</b> $j=A.length \dots$ | $\Theta(n)$ |

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**.

| (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  $b$ ,  $A[i] = a_d a_{d-1} \dots a_1$ .

RADIX-SORT( $A, d$ )

```

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 COUNTING-SORT //  $A^{j-1}$  ordinato

```

### Correttezza di Radix Sort

- **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:

1.

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

2.

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

**Complessità**

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

$m$  bit,  $r$  bit per cifra,  $\frac{m}{r}$  cifre, base  $2^r$

$$\begin{aligned}
 \Theta\left(\frac{m}{r}(m+2^r)\right) &= && r = \log_2 n \\
 &= \Theta\left(\frac{m}{\log n}(n+2^{\log n})\right) \\
 &= \Theta\left(\frac{m}{\log n}n\right) && m = O(\log n) \\
 &= \Theta(n)
 \end{aligned}$$



## 5 Tabelle Hash

$U$  universo delle chiavi

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

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

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

INSERT( $T, x$ )

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

DELETE( $T, x$ )

1  $T[x.key] = \text{NIL}$  //  $\Theta(1)$

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**

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

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

$h: U \rightarrow \{0, 1, \dots, m - 1\}$  funzione di **hashing**

$n = \#$  elementi memorizzati nella tabella  $T$

$m = \#$  celle

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

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 Elimina  $x$  da  $T[h(x.key)]$  //  $O(1)$

SEARCH( $T, k$ )

1 Cerca in  $T[h(k)]$  un elemento  $x$  tale che  $x.key = k$  //  $O(n)$

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

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

### 5.1.1 Hashing Uniforme Semplice

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

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

Consideriamo  $n_1, n_2, \dots, n_{m-1}$  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  $x_i$ :

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

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

$\alpha$  costante

$n = O(m)$

$n \leq k \cdot m$

$\alpha = \frac{n}{m} \leq k$

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

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

### 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:

- $k \in [0, 1)$  ( $0 \leq k < 1$ ),  $k$  chiave, estratta in modo indipendente dalla distribuzione uniforme (non realistica).
- Allora  $h(k) = \lfloor mk \rfloor$  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

$$\begin{aligned} U &= \{0, 1, \dots, |U| - 1\} \\ h(k) &\in \{0, 1, \dots, m - 1\} \\ h(k) &= k \bmod m \end{aligned}$$

- $m = 2^p$  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

$$k \in U$$

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

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

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

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

$$m(kA \bmod 1)$$

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

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

### 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  $h \in H$ .

Sotto certe ipotesi ottengo per **Search**:

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

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

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

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

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

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

## 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:

$$h(k, 0), h(k, 1), \dots, h(k, m-1) \quad \forall k \in U$$

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 ( $T[j] = \text{NIL}$ )
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.

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

Caratteristiche:

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

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

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

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

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

**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 ( $MCD = 1$ );
- $h(k, 0), h(k, 1), \dots, h(k, m-1)$  è permutazione di  $\{0, 1, \dots, m-1\}$ ;
- $i, i' < m \quad h(k, i) = h(k, i') \Rightarrow i = i' \quad (\text{iniettività})$

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

**iniettiva  $\Rightarrow$  biiettiva**

$$\begin{aligned} h(k, i) &= h(k, i') \\ (h_1(k) + i h_2(k)) \bmod m &= (h_1(k) + i' h_2(k)) \bmod m \\ ((i - i') h_2(k)) \bmod m &= (i h_2(k) - i' h_2(k)) \bmod m = 0 \\ (i - i') \bmod m &= 0 \\ i \geq i' \quad i - i' < m - 1 \\ \Rightarrow i - i' &= 0 \\ \Rightarrow i &= i' \end{aligned}$$

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

**es.**  $h_2(k) = 1 + h'(k \bmod m')$  con  $m' < m$ ,  $m$  primo

**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 # tentativi

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

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

(b)  $m$

**Ricerca di una chiave presente**

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

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

Costo atteso per  $k_i$ :

$$\begin{aligned} \frac{1}{1 - \alpha_{i-1}} &= \frac{1}{1 - \frac{i-1}{m}} & \left( \alpha_{i-1} = \frac{i-1}{m} < 1 \right) \\ &= \frac{m}{m - i + 1} \end{aligned}$$

Il costo atteso è la media per  $i = 1, \dots, n$

$$\begin{aligned} &= \frac{1}{n} \sum_{i=1}^n \frac{m}{m - i + 1} = \frac{m}{n} \sum_{i=1}^n \frac{1}{m - i + 1} & \left( \frac{m}{n} = \frac{1}{\alpha} \right) \\ &= \frac{1}{\alpha} \sum_{j=m-n+1}^m \frac{1}{j} \end{aligned}$$



◦ 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.

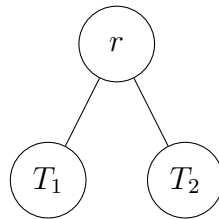
| $\alpha$        | $\frac{l}{1-\alpha}$ | $\frac{1}{\alpha} \log \left( \frac{1}{1-\alpha} \right)$ |
|-----------------|----------------------|---|
| $\alpha = 0.3$  | 1.43                 | 1.19  |
| $\alpha = 0.5$  | 2.00                 | 1.39  |
| $\alpha = 0.7$  | 3.33                 | 1.72 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

## 6 Alberi

### 6.1 Alberi Binari di Ricerca

**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.



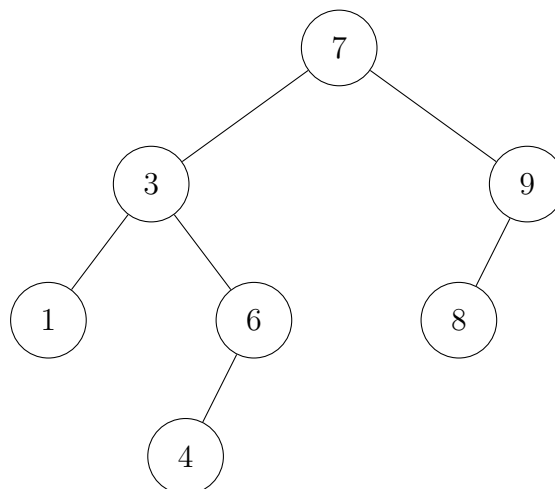
Ogni nodo  $x$  ha i seguenti campi:

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

**Proprietà**  $\forall r$

- $\rightarrow$  Per ogni nodo  $y$  in  $T_1$   $y.key \leq x.key$ ;
- $\rightarrow$  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( $x.\text{right}$ )

```

### Complessità

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

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  $n' < n$

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

### 6.1.2 Ricerca

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

- Se  $x$  è NIL  $\Rightarrow$  restituisce NIL;
- Altrimenti se  $x.\text{key} = k \Rightarrow$  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( $x.\text{right}, k$ )
```

**Complessità** La complessità è l'altezza  $h$  dell'albero, ovvero  $O(h)$ .

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  $x$ 
```

**Complessità**  $O(h)$

### 6.1.3 Successore

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  $\text{MIN}(x.\text{right})$ ;
- 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  $y$ 
```

**Complessità**  $O(h)$

#### 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          $y.right = z$ 
```

**Complessità**  $O(h)$

#### 6.1.5 Eliminazione

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 complessità  $O(1)$ .

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( $T, z, y$ )
```

**Complessità**  $O(h)$

## 6.2 Red-Black Trees

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

- RED per il rosso;
- BLACK per il nero.

**Accorgimento** NIL sarà in realtà un nodo,  $T.nil$ , con  $T.nil.color = BLACK$ .

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

- (1) Ogni nodo  $x$  ha  $x.color \in \{RED, BLACK\}$ ;
- (2) La radice  $root$  è BLACK;
- (3) Le foglie ( $T.nil$ ) sono BLACK;
- (4) Se  $x$  è RED, i figli sono BLACK;
- (5) Per ogni nodo  $x$ , ogni cammino da  $x$  a una qualsiasi delle foglie ha lo stesso numero di nodi BLACK (calcolato con  $bh(x)$ ).

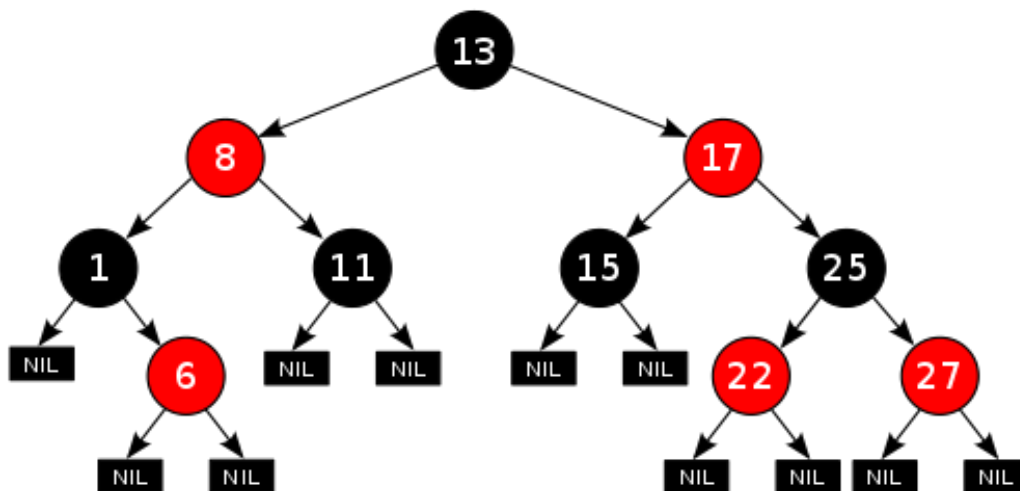


Figura 6: Esempio di un RB-tree.



È possibile notare che:

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

**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

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

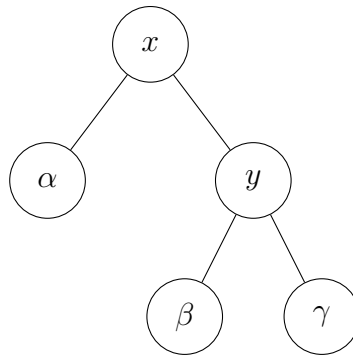
### 6.2.1 Complessità Algoritmi RB-Trees

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

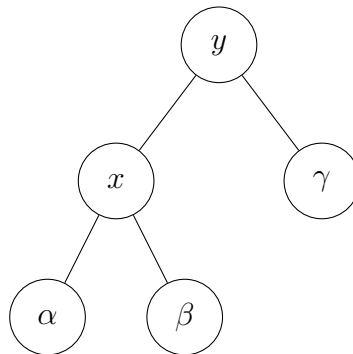
### 6.2.2 RB-Insert e RB-Delete

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  $\text{Left}(T, x)$ , ottenendo:



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

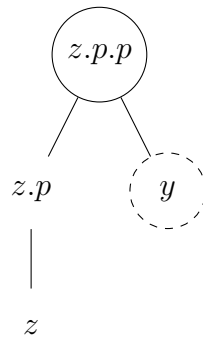
$$\alpha \rightarrow x \rightarrow \beta \rightarrow y \rightarrow \gamma$$

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

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

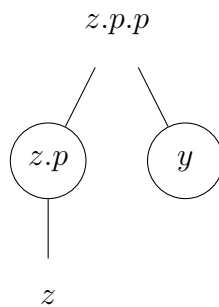
- Se violo (2)  $\Rightarrow z.color = \text{BLACK}$ ;
- Se violo (4):
  - Risolvo localmente;
  - Sposto verso l'alto il problema.

Abbiamo due **macrocasi**.  $z.p$  è figlio sinistro, oppure destro. Noi analizzeremo solo il primo:  **$z.p$  figlio sinistro**<sup>2</sup>.



Abbiamo due possibilità per  $y$ :

1.  $y.color = \text{RED}$ . Inverto il colore di  $z.p.p$  con quello dei figli, ottenendo:



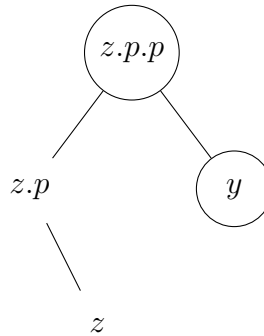
In questo modo, risolviamo localmente e rimandiamo il problema in alto.

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

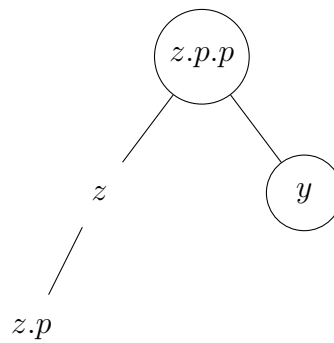
<sup>2</sup>I nodi non cerchiati sono RED, quelli cerchiati sono BLACK, quelli tratteggiati e puntinati possono essere sia RED che BLACK.

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

(2.1)  $z$  figlio destro.

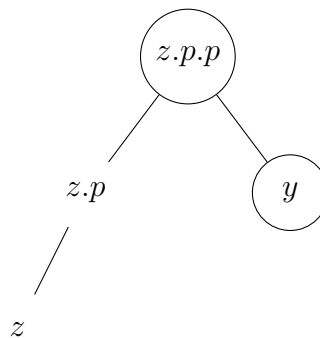


Applico  $\text{Left}(T, z.p)$ , ottenendo:

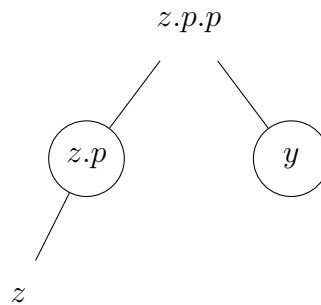


Mi riconduco al caso (2.2).

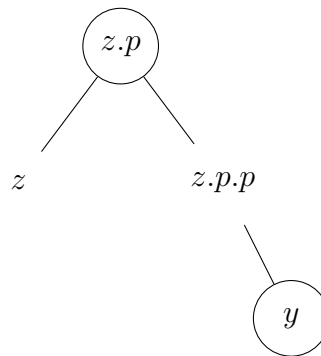
(2.2)  $z$  figlio sinistro.



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



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



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

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

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

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

---

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

RB-INSERT-FIXUP( $T, z$ )

```

1  while  $z.p.color = \text{RED}$ 
2      if  $z.p = z.p.p.left$  // Macrocaso  $z.p$  figlio sinistro
3           $y = z.p.p.right$ 
4          if  $y.color = \text{RED}$  // Caso 1
5               $z.p.p.color = \text{RED}$ 
6               $z.p.color = \text{BLACK}$ 
7               $y.color = \text{BLACK}$ 
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              $z.p.color = \text{BLACK}$ 
15              $z.p.p.color = \text{RED}$ 
16             RIGHT( $T, z.p.p$ )
17         else
18             ... // Macrocaso  $z.p$  figlio destro
19      $T.root.color = \text{BLACK}$ 

```

**Complessità**  $O(\log n) + \max 2$  rotazioni

**RB-Delete( $T, z$ )** Distinguiamo 2 casi:

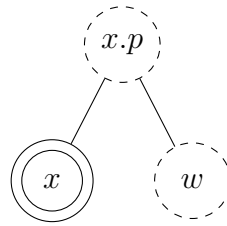
- (1)  $z$  ha un figlio;
- (2)  $z$  ha due figli.

Ci comportiamo allo stesso modo della **Delete( $T, z$ )** per un ABR, facendo però un'ulteriore accorgimento:

- se  $z.color = \text{RED}$  non ho problemi
- se  $z.color = \text{BLACK}$  violo (5):
  - Risolvo localmente;
  - Sposto verso l'alto il problema.

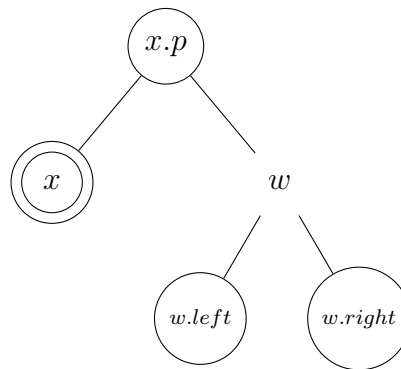
Dunque, in seguito alla **Delete( $T, z$ )**, il nodo  $x$  che ha preso il posto di  $z$  ne "assorbirà" il colore diventando doppiamente BLACK.

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

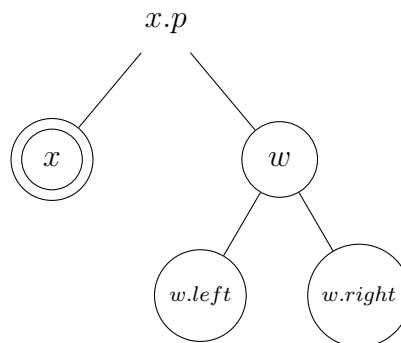


Abbiamo due possibilità per  $w$ :

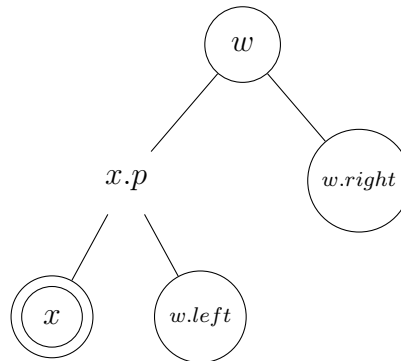
1.  $w.color = \text{RED}$



Scambio i colori di  $w$  con  $x.p$ , ottenendo:

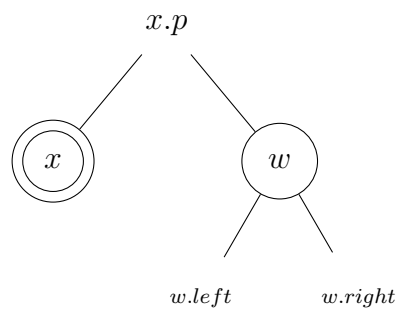


Applico  $\text{Left}(T, x.p)$ , ottenendo:



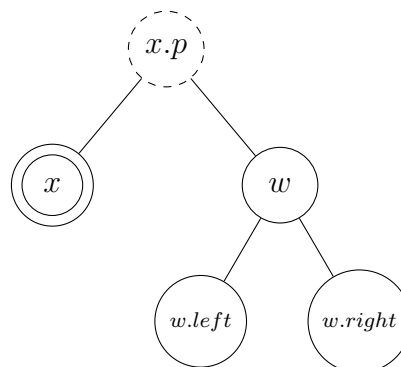
Mi riconduco al caso 2

2.  $w.color = \text{BLACK}$



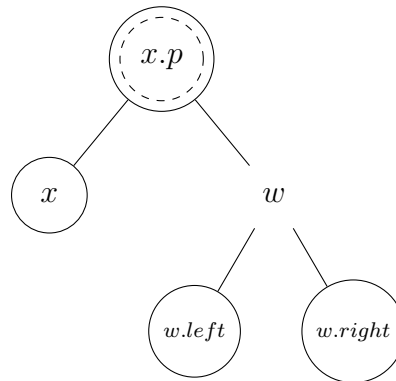
Possiamo distinguere tre sottocasi:

(2.1)  $w.left.color = \text{BLACK}$  e  $w.right.color = \text{BLACK}$



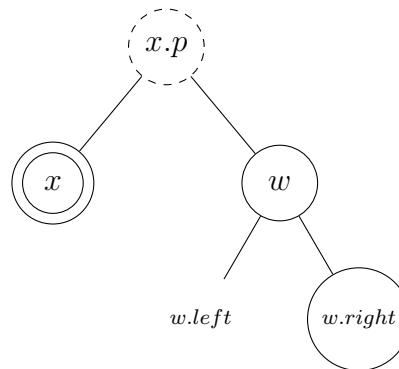
$x$  cede un suo BLACK a  $x.p$  e  $w$  diventa per forza RED, ottenendo:



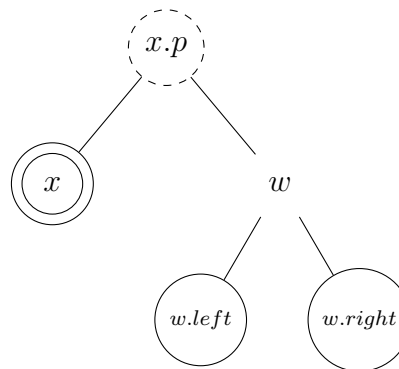


In questo modo, risolviamo localmente e rimandiamo il problema in alto.

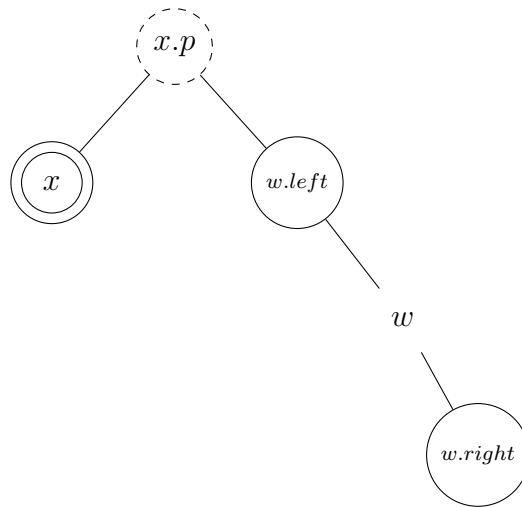
(2.2)  $w.right.color = \text{BLACK}$



Scambio i colori di  $w$  con  $w.left$ , ottenendo:

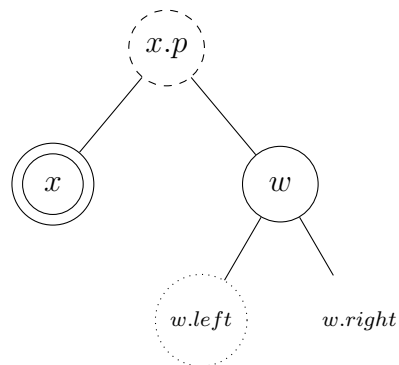


Applico  $\text{Right}(T, w)$ , ottenendo:

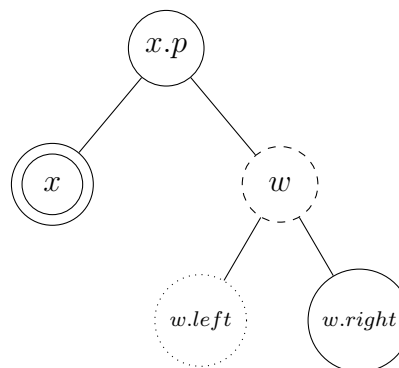


Mi riconduco al caso (2.3)

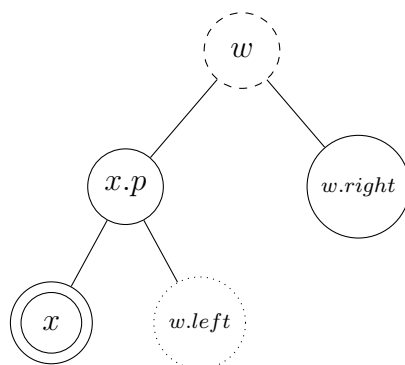
(2.3)  $w.right.color = \text{RED}$



Scambio i colori di  $x.p$  con  $w$  e  $w.right$  diventa BLACK, ottenendo:



Applico  $\text{Left}(T, x.p)$ , ottenendo:



Ho risolto.

**Complessità**  $O(\log n) + \max 3 \text{ rotazioni}$

## 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  $x$ .

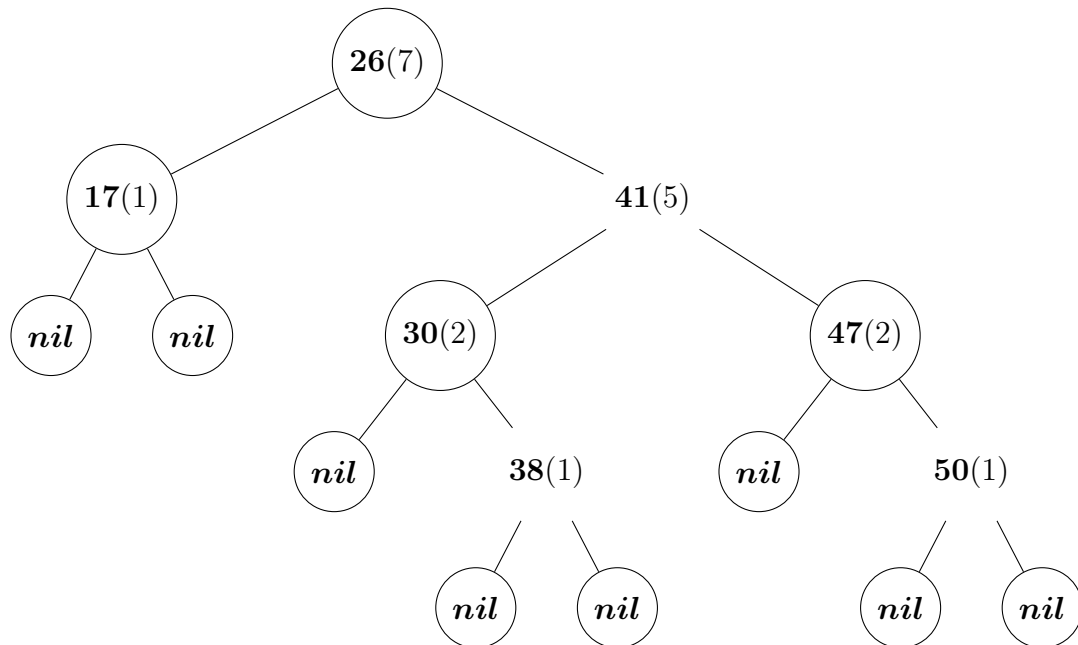
Per implementare queste due procedure, ho bisogno di un nuovo campo dati. Aggiungo il campo

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

Valgono

$$\begin{aligned} T.nil.size &= 0 \\ x.size &= x.left.size + x.right.size + 1 \end{aligned}$$

**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  $x.size$ 
  
```

**Complessità** La complessità 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  $T_x$  in una visita simmetrica.

```

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( $x.right, i - r$ )

```

**Complessità**  $O(h) = O(\log n)$

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

```

RANK( $x$ )
1   $r = x.left.size + 1$ 
2   $y = x$ 
3  while  $y \neq T.root$  // 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  $r$ 

```

**Complessità**  $O(h) = O(\log n)$

Versione aggiornata di RB-Insert :

```

RB-INSERT( $T, z$ )
    // (1) versione aggiornata di Insert
    1   $z.size = 1$  // modifica 1
    2   $x = T.root$ 
    3   $y = T.nil$ 
    4  while  $x \neq T.nil$ 
    5       $x.size = x.size + 1$  // modifica 2
    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 if  $z.key < y.key$ 
    15       $y.left = z$ 
    16      else
    17          $y.right = z$ 
    // (2) RB-Insert-Fixup
    18   $z.color = RED$ 
    19  RB-INSERT-FIXUP( $T, z$ )

```

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   $y.size = x.size$  // modifica 1
    7   $x.size = x.left.size + x.right.size + 1$  // modifica 2

```

Versione aggiornata di Delete:

```

DELETE( $T, z$ )
    // Identica a quella per gli RB-Trees
    :
1   $w = y.p$ 
2  while  $w \neq \text{NIL}$ 
3       $w.size = w.size - 1$ 
4       $w = w.p$ 

```

Versione aggiornata di RB-Delete:

```

RB-DELETE( $T, z$ )
1  DELETE( $T, z$ ) //  $O(\log n)$ 
2  RB-DELETE-FIXUP( $T, z$ ) //  $O(\log n)$ 

```

### 6.3.2 Teorema dell'Aumento degli RB-Trees

**Def.** Sia  $x.field$  un campo che si calcola in tempo costante 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

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

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

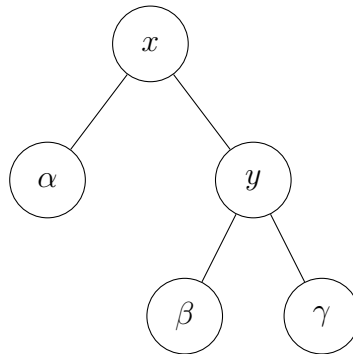
Vogliamo implementare le seguenti procedure:

- **Insert**( $T, x$ )
- **Delete**( $T, x$ )
- **ISearch**( $T, i$ ) con  $i = [low, high]$ :

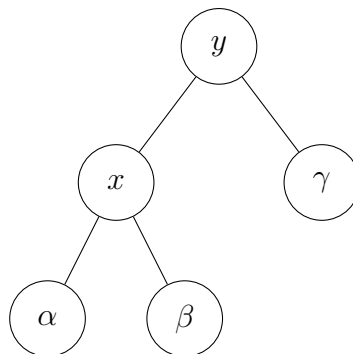


- $x$  tale che  $x.int \cap i \neq \emptyset$ ;
- $T.nil$  se un tale  $x$  non c'è.

**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 ( $x.left \neq T.nil$ ) and ( $x.left.max \geq i.low$ )
4      return ISearch( $x.left, i$ )
5  else
6      return ISearch( $x.right, i$ )
  
```

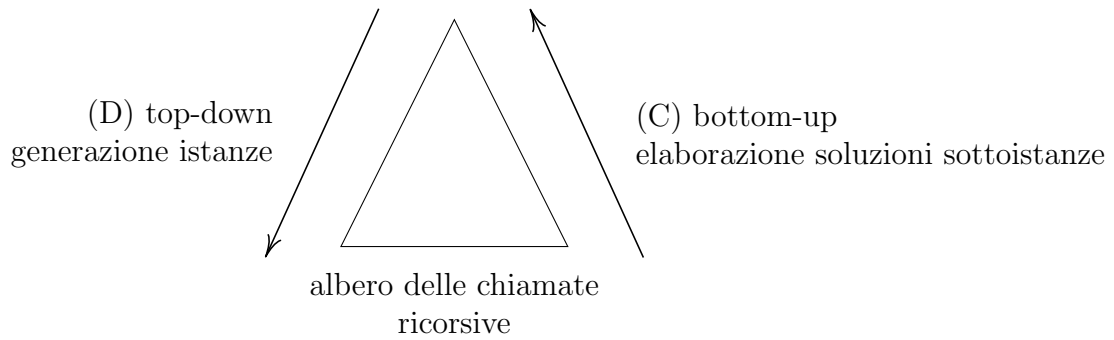
**Correttezza**

- **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.**  $\forall i' \text{ in } x.left \Rightarrow i' \cap i \neq \emptyset$ .

**Complessità**  $O(h) = O(\log n)$

## 7 Programmazione Dinamica

### 7.1 Critica al Divide & Conquer (D&C)



Il processo di soluzione non ha memoria, quindi le soluzioni di sottoistanze vanno ricalcolate.

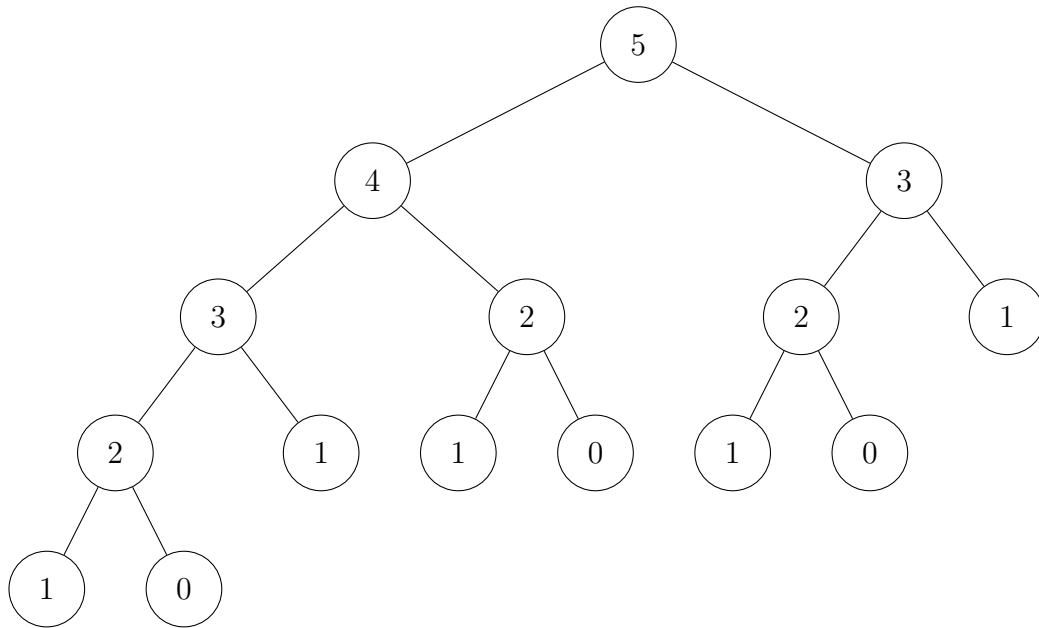
**Esempio** Vediamo uno "spreco" usando D&C: la sequenza di Fibonacci.

$$F(n) = \begin{cases} 1 & \text{se } n = 0, 1 \\ F(n-1) + F(n-2) & \text{se } n \geq 2 \end{cases}$$

REC-FIB( $n$ )

```
1  if ( $n = 0$ ) or ( $n = 1$ )
2      return 1
3  return REC-FIB( $n - 1$ ) + REC-FIB( $n - 2$ )
```

Ad esempio, con  $n = 5$



Vengono ricalcolate  $F(3)$  e  $F(2)$ .

### Complessità

$$T(n) = \begin{cases} 0 & \text{se } n = 0, 1 \quad (\text{il "return" costa } 0) \\ T(n-1) + T(n-2) + 1 & \text{se } n \geq 2 \quad (\text{il "+" costa } 1) \end{cases}$$

$$\begin{aligned} T(n) &\geq T(n-1) + T(n-2) + 1 \\ &\geq 2T(n-2) + 1 \\ &\geq 2(2T(n-2-2) + 1) + 1 \\ &= 2^2T(n-2-2) + 2 + 1 \\ &\geq 2^iT(n-2 \cdot i) + \sum_{j=0}^{i-1} 2^j \end{aligned}$$

$$i_0 \rightarrow i = \left\lfloor \frac{n}{2} \right\rfloor$$

$$\text{se } n \text{ è pari: } 2^{\frac{n}{2}}T(n-2 \cdot \frac{n}{2}) = 2^{\frac{n}{2}}T(0)$$

$$\begin{aligned} \text{se } n \text{ è dispari: } \quad &\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2} \\ &2^{\frac{n-1}{2}}T(n-2 \cdot \frac{n-1}{2}) = 2^{\frac{n-1}{2}}T(1) \end{aligned}$$

Otteniamo

$$T(n) \geq \sum_{j=0}^{\lfloor \frac{n}{2} \rfloor - 1} 2^j = \Theta(2^{\frac{n}{2}})$$

In verità,

$$T(n) = \Theta\left(\left(\frac{1 + \sqrt{5}}{2}\right)^n\right)$$

Vediamo ora una versione iterativa:

IT-FIB( $n$ )

```

1  if ( $n = 0$ ) or ( $n = 1$ )
2      return 1
3   $F[0] = 1$ 
4   $F[1] = 1$ 
5  for  $i = 2$  to  $n$ 
6       $F[i] = F[i - 1] + F[i - 2]$ 
7  return  $F[n]$ 
```

**Complessità**  $\Theta(n)$

La programmazione dinamica salta la fase top-down.

## 7.2 Memoizzazione

È un ibrido tra il D&C e la programmazione dinamica che vuole mantenere la fase top-down pur cercando di ricordare le soluzioni ai sottoproblemi

**Def** Un algoritmo **memoizzato** è costituito da due subroutine distinte:

- 1) **routine di inizializzazione**: risolve direttamente i casi base e inizializza una struttura dati che contiene le soluzioni ai casi base e gli elementi per tutte le sottoistanze da calcolare, inizializzate ad un valore di default
- 2) **routine ricorsiva**: esegue il codice D&C preceduto da un test sulla struttura dati per verificare se la soluzione è già stata calcolata e memorizzata. Se sì, si ritorna, altrimenti la si calcola ricorsivamente e la si memorizza nella struttura.

**Esempio** Riprendiamo l'esempio di prima sulla sequenza di Fibonacci.

INIT-FIB( $n$ )

```

1  if ( $n = 0$ ) or ( $n = 1$ )
2      return 1
3   $F[0] = 1$ 
4   $F[1] = 1$ 
5  for  $i = 2$  to  $n$ 
6       $F[i] = 0$ 
7  return REC-FIB( $n$ )

```

**Complessità**  $\Theta(n)$

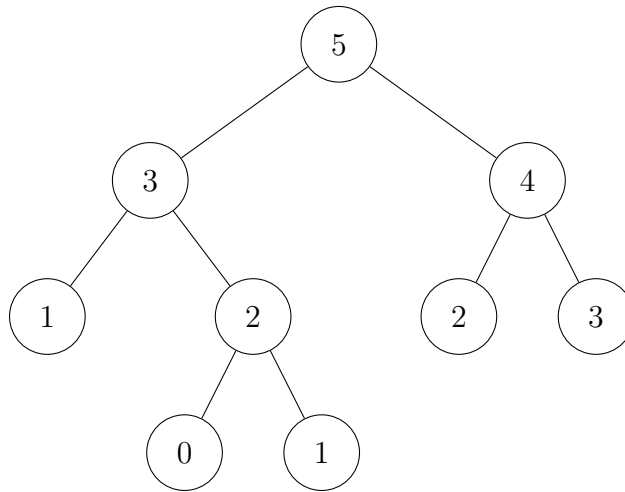
REC-FIB( $i$ )

```

1  if  $F[i] = 0$ 
2       $F[i] = \text{REC-FIB}(i - 2) + \text{REC-FIB}(i - 1)$ 
3  return  $F[i]$ 

```

Ad esempio, con  $n = 5$



Questa volta,  $F(3)$  e  $F(2)$  non vengono ricalcolate. Abbiamo  $n$  foglie e  $n - 1$  nodi interni ( $n$  parte da 0).

### 7.3 Problemi di Ottimizzazione

$I$  = insieme delle istanze

$S$  = insieme delle soluzioni

$\Pi \subseteq I \times S$

$\forall i \in I, S(i) = \{s \in S : (i, s) \in \Pi\}$  = insieme delle soluzioni ammissibili

$c: S \rightarrow \mathbb{R}$  = funzione di costo

Determinare, data  $i \in I$ ,  $s^* \in S(i) : c(s^*) = \min(/ \max)\{c(s) : s \in S(i)\}$

#### Problema della raggiungibilità su un grafo orientato

$I = \{\langle G = (V, E), u, v \rangle : V \subseteq \mathbb{N}, V \text{ finito}, E \subseteq V \times V, u, v \in V\}$

$S = \{\langle v_1, v_2, \dots, v_k \rangle : k \geq 1, v_i \in \mathbb{N} \ \forall 1 \leq i \leq k\} \cup \{\varepsilon\}$  ( $\varepsilon$  = cammino vuoto)

$(i = \langle G = (V, E), u, v \rangle, s) \in \Pi \iff \begin{cases} S = \varepsilon, \exists \text{ un cammino tra } u \text{ e } v \text{ in } G \\ S = \langle v_1, v_2, \dots, v_k \rangle, v_1 = u, v_k = v, \\ (v_i, v_{i+1}) \in E \ \forall 1 \leq i \leq k \end{cases}$

$c(\langle v_1, v_2, \dots, v_k \rangle) = k - 1$

$c(\varepsilon) = +\infty$

**Caratteristiche** Un problema di ottimizzazione, per essere risolto con la programmazione dinamica, deve avere le seguenti caratteristiche:

- struttura ricorsiva;
- esistenza di sottoistanze ripetute;
- spazio di sottoproblemi "piccolo".

#### Paradigma Generale

1. Caratterizza la struttura di una soluzione ottima  $s^*$  in funzione di soluzione ottime  $s_1^*, s_2^*, \dots, s_k^*$  di sottoistanze di taglia inferiore.
2. Determina una relazione di ricorrenza del tipo  $c(s^*) = f(c(s_1^*), \dots, c(s_k^*))$ .
3. Calcola  $c(s^*)$  impostando il calcolo in maniera bottom-up (oppure memoizzando).
4. Mantiene informazioni strutturali aggiuntive che permettono di ricostruire  $s^*$ .

## 7.4 Problemi su Stringhe

**Def** Dato un alfabeto finito  $\Sigma$ , una **stringa**

$$X = \langle x_1, x_2, \dots, x_m \rangle, \quad x_i \in \Sigma \quad \forall 1 \leq i \leq m$$

è una concetazione finita di simboli in  $\Sigma$ .

$$m = |X| = \text{lunghezza di } X$$

$$\Sigma^* = \text{insieme di tutte le stringhe di lunghezza finita costruibili su } \Sigma$$

$$\varepsilon = \text{stringa vuota}$$

Data una stringa  $X$ , il **prefisso** di  $X$  è

$$X_i = \langle x_1, x_2, \dots, x_i \rangle, \quad 1 \leq i \leq m$$

Data una stringa  $X$ , il **suffisso** di  $X$  è

$$X^i = \langle x_i, x_{i+1}, \dots, x_m \rangle, \quad 1 \leq i \leq m$$

$$\text{Per convenzione } X_0 = X^{m+1} = \varepsilon$$

**Def** Data una stringa  $X$ , la **sottostringa** di  $X$  è

$$X_{i\dots j} = \langle x_i, x_{i+1}, \dots, x_j \rangle, \quad 1 \leq i \leq j \leq m$$

$$\text{Per convenzione } X_{i\dots j} = \varepsilon \quad \text{se } i > j$$

# possibili sottostringhe di una stringa con  $m$  caratteri:

$$\begin{array}{ccccc} \binom{m}{2} & + & m & + & 1 \\ \uparrow & & \uparrow & & \uparrow \\ i \neq j & & i = j & & \varepsilon \end{array} = \frac{m(m+1)}{2} = \Theta(m^2)$$

Lo spazio delle sottostringhe "non è troppo grande".

**Def** Data una stringa

$$X = \langle x_1, x_2, \dots, x_m \rangle \in \Sigma^*$$

e

$$Z = \langle z_1, z_2, \dots, z_k \rangle \in \Sigma^*$$

si dice che  $Z$  è **sottosequenza** di  $X$  se  $\exists$  una successione crescente di indici

$$1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq m : z_j = x_{i_j} \quad \forall 1 \leq j \leq k$$



**Esempio**

$$X = \langle a, b, c, b, b, d \rangle$$

$$Z_1 = \langle a, b, c \rangle = X_{1\dots 3}$$

$$Z_2 = \langle a, c, b \rangle \quad i_1 = 1, \quad i_2 = 3, \quad i_3 = 4 \text{ o } 5$$

$$Z_3 = \langle b, b \rangle = X_{4\dots 5} \quad i_1 = 2, \quad i_2 = 5$$

# possibili sottosequenze di una stringa con  $m$  caratteri:

$$\sum_{k=0}^m \binom{m}{k} = 2^m$$

$\uparrow$   
 stringhe lunghe  $k$   
 prese da un insieme  
 di  $m$  elementi

## 7.5 Longest Common Subsequence (LCS)

Date due stringhe  $X, Y$  determina  $Z$  tale che:

- 1)  $Z$  è sottosequenza di  $X$  e  $Y$ ;
- 2)  $Z$  è la più lunga tra tutte le sottosequenze comuni.

### Esempio

$$X = \langle a, b, c, b, b, d \rangle$$

$$Y = \langle a, d, c, c, b, d \rangle$$

$Z = \langle a, c, b, d \rangle$  è una LCS (in questo caso è l'unica)

$$i_1 = 1, \quad i_2 = 3, \quad i_3 = 4 \text{ o } 5, \quad i_4 = 6$$

$$j_1 = 1, \quad j_2 = 3 \text{ o } 4, \quad j_3 = 5, \quad j_4 = 6$$

Risolvero il problema:

$$|X| = m$$

$$|Y| = n$$

L'approccio "brute force" ha complessità  $\Omega(2^m \cdot 2^n)$ .

Devo cercare di individuare una proprietà di sottostruttura, cioè la LCS deve "nascondere" al suo interno LCS di qualche stringa più piccola di  $X$  e  $Y$ .

$$X = \langle b, c, f, a \rangle$$

$$Y = \langle c, f, d, a \rangle$$

$$Z = LCS(X, Y) = \langle Z', a \rangle \quad \text{con } Z' = LCS(X_3, Y_3)$$

$$X = \langle X', a \rangle$$

$$Y = \langle Y', b \rangle$$

$Z$  o non termina con  $a$ , o non termina con  $b$

$$Z = LCS(X', Y) \text{ o } LCS(X, Y')$$

$$S = \{LCS(X_i, Y_j) : 0 \leq i \leq m, 0 \leq j \leq n\}, \quad |S| = (m+1)(n+1)$$

$$\begin{array}{cc} \uparrow & \uparrow \\ \varepsilon & \varepsilon \end{array}$$

**7.5.1 Proprietà di Sottostruttura Ottima**

Dati i prefissi

$$X_i = \langle x_1, x_2, \dots, x_i \rangle$$

$$Y_j = \langle y_1, y_2, \dots, y_j \rangle$$

$$\text{Sia } Z = \langle z_1, z_2, \dots, z_k \rangle = LCS(X_i, Y_j)$$

0. caso base: o  $i = 0$  o  $j = 0$

$$\Rightarrow Z = \varepsilon$$

1.  $i, k > 0$

se  $x_i = y_j$  allora

$$(a) \ z_k = x_i (= y_j)$$

$$(b) \ Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$$

2.  $i, j > 0$

se  $x_i \neq y_j$  allora

$Z$  è la stringa di lunghezza massima tra  $LCS(X_i, Y_{j-1})$  e  $LCS(X_{i-1}, Y_j)$

**Dimostrazione**

0. banale

1.  $x_i = y_j$

$$Z = LCS(X_i, Y_j) = \langle z_1, z_2, \dots, z_k \rangle = \langle x_{i_1}, x_{i_2}, \dots, x_{i_k} \rangle = \langle y_{j_1}, y_{j_2}, \dots, y_{j_k} \rangle$$

$$1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq i, \quad 1 \leq j_1 \leq j_2 \leq \dots \leq j_k \leq j$$

(a) Ragioniamo per assurdo

$$z_k = x_{i_k} = y_{j_k}$$

$$z_k \neq (x_i = y_j)$$

$$\Rightarrow i_k < i, \quad j_k < j$$

$$Z' = \langle Z, x_i \rangle$$

$$1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq i_{k+1} = i, \quad 1 \leq j_1 \leq j_2 \leq \dots \leq j_k \leq j_{k+1} = j$$

(b) Devo dimostrare che

$$Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$$

$$Z_{k-1} = \langle x_{i_1}, x_{i_2}, \dots, x_{i_{k-1}} \rangle = \langle y_{j_1}, y_{j_2}, \dots, y_{j_{k-1}} \rangle$$

$$i_{k-1} \leq i - 1 < i$$

$$Z_{k-1} = CS(X_{i-1}, Y_{j-1})$$

Ora dimostro che

$$Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$$

Suppongo per assurdo che

$$Z_{k-1} \neq LCS(X_{i-1}, Y_{j-1})$$

$$\Rightarrow \exists Z' \text{ con } |Z'| \geq k$$

$$\Rightarrow \text{creo } Z'' = \langle Z', x_i (= y_j) \rangle$$

$$\begin{array}{ccc} \uparrow & \uparrow & \\ \geq k & 1 & \Rightarrow \geq k+1 \end{array}$$

2. (come esercizio)

Il D&C "non funziona" perchè ci sono molti sottoproblemi ripetuti.

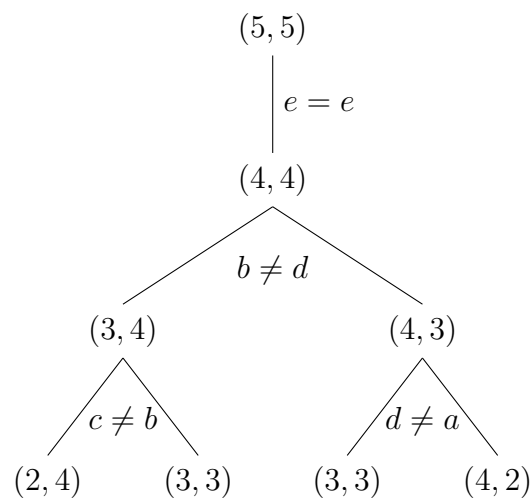
### Esempio

$$X = \langle a, b, c, d, e \rangle$$

$$Y = \langle b, c, a, b, e \rangle$$

Trova  $LCS(X, Y)$

Albero delle chiamate:



L'istanza  $(3, 3)$  è ripetuta.

**Complessità Strategia Ricorsiva** (Modello di costo: confronto tra caratteri)

$$T(n, m) = \begin{cases} 0 & \text{se } n = 0 \text{ o } m = 0 \\ T(n-1, m) + T(n, m-1) + 1 & \text{se } n, m > 0 \end{cases}$$

Si dimostra che

$$T(n, m) = \Theta \left( \binom{m}{n} \right)$$

$$\binom{m}{2} \geq \binom{m}{2}^n$$

Caso  $m = 2n$

$$\binom{m}{2}^n = 2^n$$

### 7.5.2 Ricorrenza sui Costi

La scrittura della ricorrenza sui costi è il secondo passo per costruire un algoritmo di programmazione dinamica.

Definisco

$$l(i, j) = |LCS(X_i, Y_j)|$$

$$l(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 & (\text{caso 0}) \\ l(i-1, j-1) + 1 & \text{se } i, j > 0 \text{ e } x_i = x_j & (\text{caso 1}) \\ \max\{l(i, j-1), l(i-1, j)\} & \text{se } i, j > 0 \text{ e } x_i \neq x_j & (\text{caso 2}) \end{cases}$$

Alla fine ci interessa calcolare  $l(m, n)$ .

Per calcolare  $l(i, j)$  mi possono servire tre valori:

$$L = \left[ \begin{array}{ccc} & (i-1, j-1) & (i-1, j) \\ & \swarrow & \uparrow \\ (i, j-1) & \leftarrow & (i, j) \end{array} \right]$$

Scansione "row-major": riempio la tabella per righe, da sinistra a destra.

Informazione aggiuntiva per costruire la sequenza (vera e propria):

$$b(i, j) = \begin{cases} \swarrow & \text{se } x_i = y_j \\ \leftarrow & \text{se } x_i \neq x_j \text{ e } \max = LCS(i, j-1) \\ \uparrow & \text{se } x_i \neq y_j \text{ e } \max = LCS(i-1, j) \end{cases}$$

**Pseudocodice**LCS( $X, Y$ )

```
1   $m = X.length$ 
2   $n = Y.length$ 
3  for  $i = 0$  to  $m$ 
4       $L[i, 0] = 0$ 
5  for  $j = 0$  to  $n$ 
6       $L[0, j] = 0$ 
7  for  $i = 1$  to  $m$ 
8      for  $j = 1$  to  $n$ 
9          if  $x_i = y_j$ 
10              $L[i, j] = L[i - 1, j - 1] + 1$ 
11              $B[i, j] = '\nwarrow'$ 
12          else if  $L[i - 1, j] \geq L[i, j - 1]$ 
13              $L[i, j] = L[i - 1, j]$ 
14              $B[i, j] = '\uparrow'$ 
15          else
16              $L[i, j] = L[i, j - 1]$ 
17              $B[i, j] = '\leftarrow'$ 
18  return ( $L[m, n], B$ )
```

**Complessità**

$$T(m, n) = \Theta(m \cdot n)$$

$$\text{Caso } m = n \Rightarrow T(m, n) = \Theta(n^2)$$

Procedura per stampare la LCS:

PRINT-LCS( $B, X, i, j$ )

```
1  if  $i = 0$  or  $j = 0$ 
2      return  $\varepsilon$ 
3  if  $B[i, j] = '\nwarrow'$ 
4      PRINT-LCS( $B, X, i - 1, j - 1$ )
5      PRINT( $x_i$ )
6  else if  $B[i, j] = '\leftarrow'$ 
7      PRINT-LCS( $B, X, i, j - 1$ )
8  else //  $B[i, j] = '\uparrow'$ 
9      PRINT-LCS( $B, X, i - 1, j$ )
```

**Complessità**  $\Theta(m) = \Theta(|LCS|)$

**Esercizio**
 $X = \langle b, d, c, d \rangle$ 
 $Y = \langle a, b, c, b, d \rangle$ 

 Restituisci  $LCS(X, Y)$  e  $|LCS(X, Y)|$ 

$$L = \begin{array}{c} \begin{array}{ccccc} & a & b & c & b & d \\ \begin{array}{c} b \\ d \\ c \\ d \end{array} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & \mathbf{2} & 2 \\ 0 & 0 & 1 & 2 & \mathbf{3} \end{bmatrix} \end{array} & B = \begin{array}{c} \begin{array}{ccccc} & a & b & c & b & d \\ \begin{array}{c} b \\ d \\ c \\ d \end{array} & \begin{bmatrix} \uparrow & \swarrow & \leftarrow & \swarrow & \leftarrow \\ \uparrow & \uparrow & \uparrow & \uparrow & \swarrow \\ \uparrow & \uparrow & \swarrow & \leftarrow & \uparrow \\ \uparrow & \uparrow & \uparrow & \uparrow & \swarrow \end{bmatrix} \end{array} \end{array}$$

 $LCS(X, Y) = \langle b, c, d \rangle \quad |LCS(X, Y)| = 3$ 
**Pseudocodice Memoizzato**
 INIT-LCS( $X, Y$ )

```

1   $m = X.length$ 
2   $n = Y.length$ 
3  if ( $m = 0$ ) or ( $n = 0$ )
4      return 0
5  for  $i = 0$  to  $m$ 
6       $L[i, 0] = 0$ 
7  for  $j = 0$  to  $n$ 
8       $L[0, j] = 0$ 
9  for  $i = 1$  to  $m$ 
10     for  $i = 1$  to  $n$ 
11          $L[i, j] = -1$ 
12 return R-LCS( $X, Y, m, n$ )
  
```

 R-LCS( $X, Y, i, j$ )

```

1  if  $L[i, j] = -1$ 
2      if  $x_i = y_j$ 
3           $L[i, j] = \text{R-LCS}(X, Y, i - 1, j - 1)$ 
4      else if  $\text{R-LCS}(X, Y, i - 1, j) \geq \text{R-LCS}(X, Y, i, j - 1)$ 
5           $L[i, j] = L[i - 1, j]$ 
6      else
7           $L[i, j] = L[i, j - 1]$ 
8  return  $L[i, j]$ 
  
```



**Complessità**  $O(m \cdot n)$

**Osservazione** Se  $x_i = y_j$  sempre, invoco  $\text{R-LCS}(X, Y, i-1, j-1)$  ma non invoco mai  $\text{R-LCS}(X, Y, i-1, j)$  o  $\text{R-LCS}(X, Y, i, j-1)$ .

Ad esempio

$X = \langle a, a, b, b, c \rangle$

$Y = \langle b, b, c \rangle$

Albero delle chiamate:

$$\begin{array}{c}
 (5, 3) \\
 \left| \begin{array}{l} c = c \end{array} \right. \\
 (4, 2) \\
 \left| \begin{array}{l} b = b \end{array} \right. \\
 (3, 1) \\
 \left| \begin{array}{l} b = b \end{array} \right. \\
 (2, 0)
 \end{array}$$

In generale, se  $Y$  è suffisso di  $n \leq m$  caratteri di  $X$ , la complessità di  $\text{R-LCS}$  nel caso migliore è:

$$T_{\text{R-LCS}}(m, n) = n$$

Inoltre,

$$\begin{aligned}
 \Omega_{\text{LCS}}(m + n) &\cong \Omega(n) \\
 O_{\text{LCS}, \text{R-LCS}}(m \cdot n) &\cong O(n^2)
 \end{aligned}$$

**Spazio**

$$S_{\text{LCS}}(m, n) = \Theta(m, n)$$

Tuttavia, posso migliorarlo a

$$\Theta(2n) = \Theta(n)$$

poichè mi bastano due righe della tabella in memoria ad ogni istante, quindi due vettori lunghi  $n$ .

Inoltre, se  $m \ll n$ , posso fare un'ulteriore ottimizzazione utilizzando la tecnica "column-major", cioè scansione per colonne, con due vettori lunghi  $m$ .

## 7.6 Longest Increasing Subsequence (LIS)

**Def** Dato un alfabeto  $\Sigma$  totalmente ordinato ( $\forall a, b \in \Sigma \quad a < b \text{ o } a = b \text{ o } a > b$ ) e dato  $X = \langle x_1, x_2, \dots, x_n \rangle$ , si dice che  $Z = \langle z_1, z_2, \dots, z_k \rangle$  è **sottosequenza crescente** di  $X$  ( $Z = IS(X)$ ).

**Problema di ottimizzazione** Determinare la **più lunga sottosequenza crescente** di  $X$  ( $Z = LIS(X)$ )

**Esempio**

$$X = \langle 5, 2, 4, 3, 7, 8 \rangle$$

$$Z = LIS(X) = \langle 2, 3, 7, 8 \rangle$$

$$Z' = LIS(X) = \langle 2, 4, 7, 8 \rangle$$

**Tentativo** Data  $X$ , calcolo  $LIS(X_i) \forall 0 \leq i \leq n$

$$Z = \langle z_1, z_2, \dots, z_k \rangle = LIS(X_i)$$

o caso fortunato:  $z_k < X_{i+1}$

o caso sfortunato:  $z_k \geq X_{i+1}$

**Def**  $Z = \overline{LIS}(X_i)$  è la più lunga tra le  $IS(X_i)$  con  $Z = \langle z_1, z_2, \dots, z_k \rangle = \langle x_{i_1}, x_{i_2}, \dots, x_{i_k} \rangle$  con  $i_k = i$ .

**Esempio**

$$X = \langle 8, 2, 5, 1, 3 \rangle$$

$$LIS(X_4) = \langle 2, 5 \rangle$$

$$\overline{LIS}(X_4) = \langle 1 \rangle$$

In generale,  $LIS$  e  $\overline{LIS}$  sono problemi molto diversi.

**Osservazione**  $|LIS(X)| = \max_{1 \leq i \leq n} \{|\overline{LIS}(X_i)|\}$

**Dimostrazione**

$$\circ |LIS(X)| \leq \max_{1 \leq i \leq n} \{|\overline{LIS}(X_i)|\}$$

$$Z = LIS(X) = \langle x_{i_1}, x_{i_2}, \dots, x_{i_k} \rangle$$

$$Z = \overline{LIS}(X_{i_k})$$

$$\Rightarrow |Z| \leq \max_{1 \leq i \leq n} \{|\overline{LIS}(X_i)|\}$$

$$\circ |LIS(X)| \geq \max_{1 \leq i \leq n} \{|\overline{LIS}(X_i)|\}$$

Si dimostra analogamente al punto precedente.

### 7.6.1 Proprietà di Sottostruttura Ottima

1. caso base:  $\overline{LIS}(X_1) = \langle x_1 \rangle (= LIS(X_1))$

2.  $i > 1$

$$(a) \forall j : 1 \leq j \leq i \quad x_j \geq x_i$$

$$\overline{LIS}(X_i) = \langle x_i \rangle (= LIS(X_i))$$

$$(b) \exists \bar{j} : 1 \leq \bar{j} \leq i, \quad x_{\bar{j}} < x_i$$

$$|\overline{LIS}(X_i)| \geq 2$$

$$\overline{LIS}(X_i) = \langle z_1, z_2, \dots, z_k \rangle = \langle Z_{k-1}, x_i \rangle \text{ con } Z_{k-1} : |Z_{k-1}| = \max_{1 \leq j < i} \{|\overline{LIS}(X_j)| : x_j < x_i\}$$

### Dimostrazione

1. banale

2.  $i > 1$

$$(a) \forall j < i \quad x_j < x_i$$

$$\langle x_i \rangle = LIS(X_i) \text{ e chiaramente non può essere } |Z| > 1$$

Suppongo per assurdo che

$$|Z| = |\overline{LIS}(X_i)| > 1$$

$$\Rightarrow Z = \langle z_1, z_2, \dots, z_k \rangle, \quad k > 1$$

$$Z = \langle x_{i_1}, x_{i_2}, \dots, x_{i_k} \rangle$$

$$i_k = i$$

$$\Rightarrow i_{k-1} < i_k = i$$

Assurdo perchè allora avrei

$$x_{i_{k-1}} \geq x_{i_k}$$

che contraddice l'ipotesi che  $Z$  è una sequenza crescente!

(b) Si dimostra analogamente al sottocaso precedente.

**7.6.2 Ricorrenza sui Costi**

Definisco

$$l(i) = |\overline{LIS}(X_i)|$$

$$l(i) = \begin{cases} 1 & \text{se } i = 1 \\ 1 + \max_{1 \leq j < i} \{l(j) : x_j < x_i\} & \text{se } i > 1 \end{cases}$$

Per costruire la sottosequenza

$$\overline{LIS}(X_i) = \begin{cases} \langle x_i \rangle & (1) \\ \langle \overline{LIS}(X_{\bar{j}}), x_i \rangle & \text{con } 1 \leq \bar{j} < i \quad (2) \end{cases}$$

l'informazione addizionale è:

- $prev(i) = \begin{cases} 0 & \text{nel caso (1)} \\ \bar{j} & \text{nel caso (2)} \end{cases}$
- $len = \max_{1 \leq i \leq n} \{l(i)\}$
- $end = i$  se  $\overline{LIS}(X_i) = LIS(X)$   
(mantiene l'indice dell'ultimo carattere della LIS)

**Pseudocodice bottom-up**

$LIS(X)$

```

1  L[1] = 1
2  len = 1
3  end = 1
4  prev[1] = 0
5  for i = 2 to n
6      L[i] = 1
7      prev[i] = 0
8      for j = 1 to i - 1
9          if x_j < x_i
10             if L[i] < 1 + L[j]
11                 L[i] = 1 + L[j]
12                 prev[i] = j
13     if len < L[i]
14         len = L[i]
15         end = i
16  return (len, prev, lenend)
```

R-PRINT( $X, prev, i$ )

```
1  if  $prev[i] \neq 0$   
2      R-PRINT( $X, prev, prev[i]$ )  
3  PRINT( $x_i$ )
```

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

## 7.7 Completamento a Palindromo (CP)

**Def** Una stringa  $Z = \langle z_1, z_2, \dots, z_m \rangle$  è **palindroma** se  $z_{1+h} = z_{m-h} \quad \forall 0 \leq h \leq m-1$ .

**Problema** Data  $X = \langle x_1, x_2, \dots, x_n \rangle$ , un **complemento palindromo** di  $X$  è una stringa  $Z = CP(X) = \langle z_1, z_2, \dots, z_m \rangle$  con  $m \geq n$  tale che:

- 1)  $Z$  è palindroma;
- 2)  $X$  è sottosequenza di  $Z$

(cioè,  $Z$  è un palindromo che contiene  $X$  come sottosequenza).

### Esempio

$$X = \langle a, c, b \rangle$$

$$Z = \langle a, c, b, b, c, a \rangle$$

$$Z' = \langle a, b, c, c, b, a \rangle$$

**Osservazione**  $|X| = n \leq |Z| \leq 2n = 2|X|$

**Problema di ottimizzazione** Determinare  $Z = CP(X)$  di lunghezza minima.

Spazio dei sottoproblemi:  $X_{i\dots j}$  (cioè lo spazio delle sottostringhe di  $X$ ).

Determinare un algoritmo bottom-up quadratico nella lunghezza di  $X$ .  
 $\forall i, j : 1 \leq i \leq j \leq n$ , determinare il minimo  $Z = CP(X_{i\dots j})$ .

### 7.7.1 Proprietà di Sottostruttura Ottima

Casi base:

1.  $i = j$  (1 carattere)

$$X = \langle x_i \rangle$$

$$CP(X_{i\dots j}) = X_{i\dots j}$$

$$|CP(X_{i\dots j})| = |X_{i\dots j}|$$

2.  $j = i + 1$  (2 caratteri)

$$X_{i\dots j} = \langle x_i, x_{i+1} \rangle$$

$$(a) \ x_i = x_{i+1}$$

$$CP(X_{i...i+1}) = X_{i...i+1}$$

$$|CP(X_{i...i+1})| = 2$$

$$(b) \ x_i \neq x_{i+1}$$

Un possibile  $CP(X_{i...i+1})$  è  $\langle x_i, x_{i+1}, x_i \rangle$

$$|CP(X_{i...i+1})| = 3$$

Caso generale:

$$X_{i...j} = \langle x_i, x_{i+1}, \dots, x_j \rangle$$

$$(a) \ x_i = x_j$$

$$Z = CP(X_{i...j})$$

$$z_1 = z_k = x_i (= x_j)$$

$$Z_{2...k-1} = CP(X_{i+1...j-1})$$

(b)  $x_i \neq x_j$  Ci sono due possibili soluzioni:

$$i. \ z_1 = z_k = x_i \text{ e } Z_{2...k-1} = CP(X_{i+1...j})$$

$$ii. \ z_1 = z_k = x_j \text{ e } Z_{2...k-1} = CP(X_{i...j-1})$$

### Dimostrazione

(b).i. Suppongo per assurdo che

$$z_1 = z_k \neq x_i$$

$\Rightarrow X_{i...j}$  è sottosequenza di

$Z_{2...k-1}$  che è palindroma e più corta di 2 rispetto a  $Z$

Assurdo perchè  $Z$  è la più corta!

(b).ii. Si dimostra analogamente al sottocaso precedente.

### 7.7.2 Ricorrenza sulle Lunghezze

Definisco

$$l(i, j) = |CP(X_{i...j})|$$

$$l(i, j) = \begin{cases} 1 & \text{se } i = j \\ 2 & \text{se } j = i + 1 \text{ e } x_i = x_j \\ 3 & \text{se } j = i + 1 \text{ e } x_i \neq x_j \\ 2 + l(i + 1, j - 1) & \text{se } j > i + 1 \text{ e } x_i = x_j \\ 2 + \min\{l(i + 1, j), l(i, j - 1)\} & \text{se } j > i + 1 \text{ e } x_i \neq x_j \end{cases}$$

Per calcolare  $l(i, j)$  mi possono servire tre valori:

$$L = \left[ \begin{array}{ccc} & (i, j-1) & \leftarrow (i, j) \\ & \swarrow & \downarrow \\ (i+1, j-1) & & (i+1, j) \end{array} \right]$$

Riempio la tabella per diagonali, dall'alto verso il basso e da sinistra verso destra.

SPC( $X$ )

```

1   $n = X.length$ 
2  for  $i = 1$  to  $n - 1$ 
3       $L[i, j] = 1$ 
4      if  $x_i = x_{i+1}$ 
5           $L[i, i + 1] = 2$ 
6      else
7           $L[i, i + 1] = 3$ 
8   $L[n, n] = 1$ 
9  for  $l = 3$  to  $n$  // scansione diagonale con  $l$  indice della diagonale
10     for  $i = 1$  to  $n - l + 1$ 
11          $j = i + l - 1$ 
12         if  $x_i = x_j$ 
13              $L[i, j] = 2 + L[i + 1, j - 1]$ 
14         else
15              $L[i, j] = 2 + \min\{L[i + 1, j], L[i, j - 1]\}$ 
16 return  $L[1, n]$ 
```

**Complessità**

$$\begin{aligned}
 & \sum_{i=1}^{n-1} 1 + \sum_{l=3}^n \sum_{i=1}^{n-l+1} 1 = \sum_{i=1}^{n-1} 1 + \sum_{l=3}^n (n-l+1) = \sum_{i=1}^{n-1} 1 + \sum_{i=2}^{n-1} (n-i) = \\
 & = \sum_{i=1}^{n-1} (n-i) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2)
 \end{aligned}$$