

# Algoritmi e Strutture di Dati

37635 - 12 CFU

⚠️ ⚠️ ⚠️ Disclaimer: ⚠️ ⚠️ ⚠️

questi appunti sono un mix, sono una versione più discorsiva delle slide distribuite dai prof. Zavattaro e Di Lena tra gli A.A. 2023 e 2025

(agg. a luglio 2025) Sono gratuitamente pubblicati su  
<https://risorse.vercel.app/algoritmi-e-strutture-di-dati/appunti?from=informatica>

Se trovate errori fatemelo pure sapere scrivendo a  
✉️ [alberto.zuccari@studio.unibo.it](mailto:alberto.zuccari@studio.unibo.it) oppure su  
↗️ telegram [@b3rt0nes](https://t.me/b3rt0nes)

## SOMMARIO

---

|      |   |    |
|------|---|----|
| 1    | Introduzione agli algoritmi .....       | 4  |
| 2    | Notazione asintotica.....               | 5  |
| 2.1  | Funzione di costo.....                  | 5  |
| 2.2  | Ordini di crescita.....                 | 11 |
| 2.3  | Complessità computazionale.....         | 11 |
| 2.4  | Equazioni di ricorrenza .....           | 14 |
| 3    | Algoritmi di ordinamento .....          | 16 |
| 3.1  | Introduzione .....                      | 16 |
| 3.2  | La struttura dati array.....            | 16 |
| 3.3  | Incrementali.....                       | 17 |
| 3.4  | Divide et impera .....                  | 19 |
| 3.5  | Non-comparativi.....                    | 22 |
| 3.6  | Riassunto .....                         | 23 |
| 4    | Strutture dati elementari.....          | 24 |
| 4.2  | Strutture dati elementari.....          | 25 |
| 5    | Alberi .....                            | 27 |
| 5.2  | Alberi binari di ricerca .....          | 29 |
| 5.3  | Alberi AVL (Adelson-Velsky, Landis) ... | 34 |
| 6    | Tabelle Hash .....                      | 43 |
| 6.2  | Tabelle Hash .....                      | 43 |
| 6.3  | Funzioni hash.....                      | 44 |
| 6.4  | Risoluzione delle collisioni.....       | 45 |
| 7    | Heap e sue applicazioni .....           | 49 |
| 7.1  | Heap binari.....                        | 49 |
| 7.2  | Code con priorità .....                 | 51 |
| 8    | Union-Find .....                        | 54 |
| 8.1  | QuickFind .....                         | 54 |
| 8.2  | QuickUnion .....                        | 55 |
| 8.3  | Riepilogo .....                         | 55 |
| 8.4  | Euristiche di ottimizzazione .....      | 55 |
| 9    | Tecniche algoritmiche.....              | 57 |
| 9.1  | Divide-et-impera .....                  | 57 |
| 9.2  | Algoritmi greedy .....                  | 61 |
| 9.3  | Programmazione dinamica .....           | 64 |
| 10   | Grafi .....                             | 71 |
| 10.2 | Definizioni .....                       | 71 |
| 10.3 | Funzioni,.....                          | 75 |
| 10.4 | Implementazioni .....                   | 76 |
| 10.5 | Visite grafi .....                      | 78 |
| 10.6 | Minimum spanning tree .....             | 82 |
| 10.7 | Cammini minimi.....                     | 85 |
| 11   | Teoria della NP-Completezza .....       | 92 |
| 11.1 | Classi di complessità .....             | 92 |
| 11.2 | NP.....                                 | 94 |

# 1 INTRODUZIONE AGLI ALGORITMI

Cos'è un **algoritmo**?

Consideriamo una procedura con un numero *finito* di istruzioni per risolvere un problema

Un **algoritmo** è una **descrizione di alto livello** di una procedura.

- Esistono da prima che esistessero i computer
- Non può essere eseguito da un computer
- Può assumere un quantitativo *illimitato* di memoria

Un **programma** è l'**implementazione** di un algoritmo

- Deve essere scritto in qualche linguaggio di programmazione
- Può essere eseguito su un computer
- Deve tenere conto dei limiti di memoria del computer

Un algoritmo può prendere in *input* alcuni valori (ma può anche non farlo), segue una sequenza *finita* di *istruzioni* e può eventualmente produrre un *output*

Un algoritmo rappresenta la soluzione a un **problema** da risolvere.

- La definizione del problema vincola input e output
  - Ci sono infiniti set di istruzioni che risolvono lo stesso problema
    - Esattamente lo stesso output per lo stesso input
- (Ricetta Biscotti per 5 persone)  
(biscotti al cioccolato,  
biscotti alla crema, ...)  
(50g di burro → biscotti al burro)

Esempio:

Calcolare il Massimo Comune Divisore (MCD) tra  $x$  e  $y$  ( $x \geq y$ )

1. Dividi  $x$  per  $y$  e chiama  $r$  il resto della divisione
2. Se  $r = 0$  allora  $y$  è il MCD tra  $x$  e  $y$
3. Se  $r \neq 0$  allora assegna  $y$  ad  $x$ ,  $r$  ad  $y$  e ritorna al punto 1.

```

1. function MCD (int x, int y) -> int
2.   if y == 0 then
3.     return x
4.   else
5.     r = x mod y
6.     return MCD(y, r)

```

Algoritmo di Euclide in *pseudo-codice*:

linguaggio umano sulla sinistra;      linguaggio simile a un linguaggio di programmazione sulla destra.

Per poter sviluppare algoritmi dobbiamo:

- **Capire** il problema che vogliamo risolvere
  - Quali sono gli input e output possibili?
  - Come vengono mappati gli input sugli output?
  - Ci sono proprietà matematiche legate al nostro problema?
- **Studiare** tecniche algoritmiche e strutture dati note
  - Problemi differenti spesso condividono la stessa struttura di base
  - Molti problemi possono essere risolti modificando algoritmi noti.
- **Sapere** come stimare l'efficienza di un algoritmo
  - Tempo: quanto è veloce un algoritmo?
  - Memoria: di quanta memoria ha bisogno?
  - Come scegliere l'algoritmo migliore?

## 2 NOTAZIONE ASINTOTICA

**Scopo:** analizzare il tempo di calcolo e l'occupazione di memoria degli algoritmi in termini di *dimensione dell'input*.

Quale può essere una buona misura per il tempo di calcolo e memoria?

Tempo       $\text{sec}$   
 Memoria     $\text{Mbyte}$

Meglio considerare il *comportamento asintotico* degli algoritmi.

**Comportamento asintotico** di un algoritmo:

- Ignora costanti additive o moltiplicative e termini di ordine inferiore
- Descrive quanto crescono velocemente tempo e memoria rispetto alla dimensione dell'input
- Ci permette di confrontare le prestazioni di algoritmi differenti che risolvono lo stesso problema, indipendentemente dall'hardware su cui sono eseguiti.

### 2.1 FUNZIONE DI COSTO

**Funzione di costo:**

Dato  $n \geq 0$ , indichiamo con  $f(n) \geq 0$  la quantità di risorse (tempo o memoria) richiesta da un algoritmo in funzione dell'input di dimensione  $n$ .

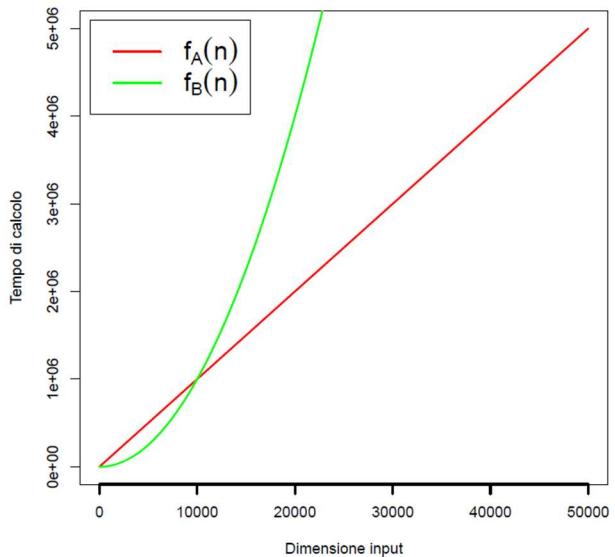
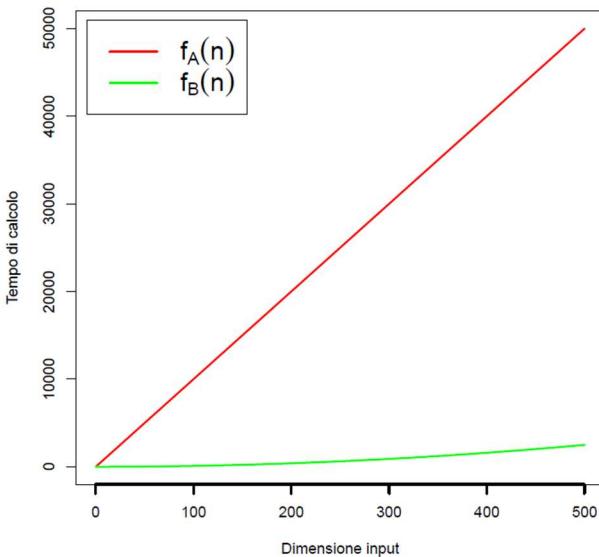
Noi siamo interessati a valutare il *rate di crescita* di  $f(n)$  ignorando fattori costanti e i termini di ordine inferiore.

Esempio:

Consideriamo due algoritmi  $A$  e  $B$  per uno stesso problema.

- $f_A(n) = 10^2 n$  è la funzione di costo del tempo di calcolo in  $A$
- $f_B(n) = 10^{-2} n^2$  è la funzione di costo del tempo di calcolo in  $B$

Quale dei due algoritmi è il più veloce?

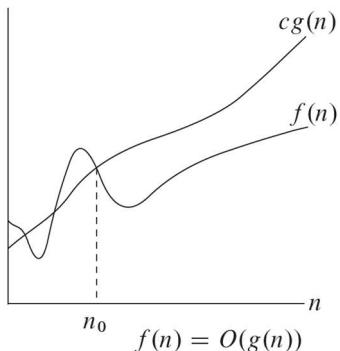


### 2.1.1 O-grande

#### O-grande

Data una funzione di costo  $g(n)$  definiamo l'insieme di funzioni per cui  $g(n)$  rappresenta un *limite asintotico superiore* come

$$O(g(n)) = \{f(n) \mid \exists c > 0, n_0 \geq 0 \text{ t.c. } \forall n \geq n_0, f(n) \leq cg(n)\}$$



- › Intuitivamente, con  $O(g(n))$  indichiamo l'insieme delle funzioni con coordinate di crescita **inferiore o uguale a  $g(n)$**
- ›  $g(n) = O(g(n))$
- › Con abuso di notazione, diciamo che  $f(n) = O(g(n))$  mentre la notazione corretta sarebbe  $f(n) \in O(g(n))$

Esempio:

Siano  $\begin{cases} g(n) = n^2 \\ f(n) = 3n^2 + 10n \end{cases}$

Dimostriamo che  $f(n) = O(g(n))$

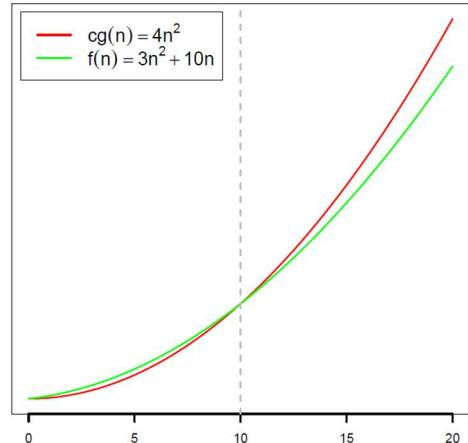
Dobbiamo trovare due costanti  $c > 0 \wedge n_0 \geq 0$  tali che:

$$\forall n \geq n_0, f(n) \leq cg(n) \Rightarrow 3n^2 + 10n \leq cn^2$$

La costante  $c$  deve soddisfare la seguente disequazione

$$c \geq \frac{3n^2 + 10}{n^2} = 3 + \frac{10}{n}$$

Verificata  $\forall c \geq 13 \wedge \forall n_0 \geq 1$

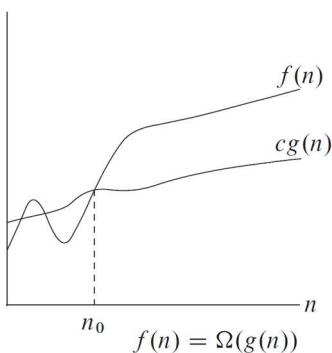


### 2.1.2 Omega-grande

#### **Ω-grande**

Data una funzione di costo  $g(n)$  definiamo l'insieme di funzioni per cui  $g(n)$  rappresenta un limite asintotico inferiore come

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 > 0 \text{ t.c. } \forall n \geq n_0, f(n) \geq cg(n)\}$$



- › Intuitivamente, con  $\Omega(g(n))$  indichiamo l'insieme di funzioni con ordine di crescita superiore o uguale a  $g(n)$

$$\rightarrow g(n) = \Omega(g(n))$$

Esempio:

Siano  $\begin{cases} g(n) = n^2 \\ f(n) = n^3 + 2n^2 \end{cases}$

Dimostriamo che  $f(n) = \Omega(g(n))$

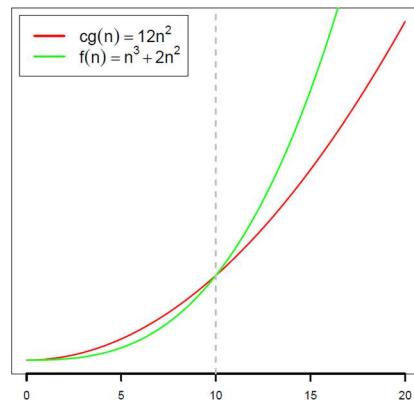
Dobbiamo cercare due costanti  $c > 0 \wedge n_0 \geq 0$  tali che:

$$\forall n \geq n_0, f(n) \geq cg(n) \Rightarrow n^3 + 2n^2 \geq cn^2$$

La costante  $c$  deve soddisfare la seguente disequazione

$$c \leq \frac{n^3 + 2n^2}{n^2} = n + 2$$

Verificata  $\forall 0 < c \leq 2, \forall n_0 \geq 0$

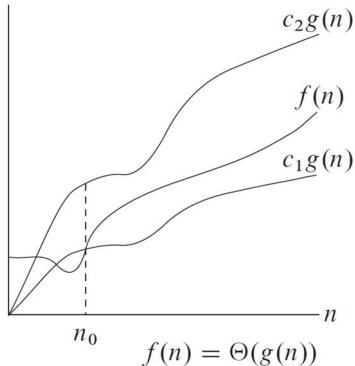


### 2.1.3 Theta

Θ

Data una funzione di costo  $g(n)$  definiamo l'insieme di funzioni *asintoticamente equivalenti* a  $g(n)$  come

$$\Theta(g(n)) = \{ f(n) \mid \exists c_1, c_2 > 0, n_0 \geq 0, \text{ t.c. } \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$



› Intuitivamente, con  $\Theta(g(n))$  indichiamo l'insieme di funzioni il cui ordine di crescita è uguale a quello di  $g(n)$

›  $g(n) = \Theta(g(n))$

#### Teorema

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

Esempio:

Siano  $\begin{cases} g(n) = n^3 \\ f(n) = n^3 + 2n^2 \end{cases}$

Dimostriamo che  $f(n) = \Theta(g(n))$

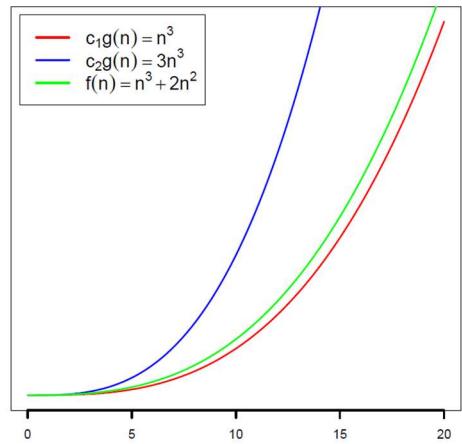
Dobbiamo cercare tre costanti,  $c_1 > 0, c_2 > 0$ , e  $n_0 \geq 0$  tali che

$$\forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \Rightarrow c_1 n^3 \leq n^3 + 2n^2 \leq c_2 n^3$$

Le costanti  $c_1, c_2$  devono soddisfare le seguenti disequazioni

$$c_1 \leq \frac{n^3 + 2n^2}{n^3} = 1 + \frac{2}{n} \leq c_2$$

Verificata:  $\begin{cases} \forall c_1 \leq 1 \\ \forall c_2 \geq 3 \end{cases}$  e  $\forall n_0 \geq 1$



### 2.1.4 o-piccolo

#### o-piccolo

Data una funzione di costo  $g(n)$  definiamo l'insieme di funzioni che sono *dominate asintoticamente* da  $g(n)$  come

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ t.c. } \forall n \geq n_0, f(n) < cg(n)\}$$

? In cosa differisce dalla notazione  $O$ -grande? ?

- ›  $f(n) = O(g(n)) \Rightarrow f(n) \leq cg(n)$  per qualche costante  $c > 0$
- ›  $f(n) = o(g(n)) \Rightarrow f(n) < cg(n)$  per tutte le costanti  $c > 0$

### 2.1.5 omega-piccolo

#### omega-piccolo

Data una funzione di costo  $g(n)$  definiamo l'insieme di funzioni che *dominano asintoticamente*  $g(n)$  come

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ t.c. } \forall n \geq n_0, f(n) < cg(n)\}$$

? In cosa differisce dalla notazione  $\Omega$ -grande? ?

- ›  $f(n) = \Omega(g(n)) \Rightarrow f(n) \geq cg(n)$  per qualche costante  $c > 0$
- ›  $f(n) = \omega(g(n)) \Rightarrow f(n) > cg(n)$  per tutte le costanti  $c > 0$

### 2.1.6 Limiti

L'ordine di crescita può essere confrontato usando i *limiti*

- › Se  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0, \Rightarrow f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$
- › Se  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty, \Rightarrow f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$
- › Se  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k > 0, \Rightarrow f(n) = \Theta(g(n))$

Intuitivamente:

| Funzioni              | Numeri reali |
|-----------------------|--------------|
| $f(n) = O(g(n))$      | $f \leq g$   |
| $f(n) = o(g(n))$      | $f < g$      |
| $f(n) = \Omega(g(n))$ | $f \geq g$   |
| $f(n) = \omega(g(n))$ | $f > g$      |
| $f(n) = \Theta(g(n))$ | $f = g$      |

A differenza di quanto avviene nel dominio dei numeri reali, non tutti gli ordini di crescita asintotica sono confrontabili

Se  $a, b \in \mathbb{R}$ , solo una tra  $a < b, a = b, a > b$  è vera

Diversamente,  $f(n) = n, g(n) = n^{\sin(n)+1}$  non sono confrontabili

- $f(n) \neq O(g(n))$  poiché quando  $\sin(n) = -1 \Rightarrow g(n) = 1$
- $f(n) \neq \Omega(g(n))$  poiché quando  $\sin(n) = 1 \Rightarrow g(n) = n^2$

### 2.1.7 Proprietà

#### Transitività

$$f(n) = O(g(n)) \text{ e } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

Vale anche per  $\Omega$ ,  $\Theta$ ,  $o$  e  $\omega$

#### Riflessività

$$f(n) = O(f(n))$$

Vale anche per  $\Omega$  e  $\Theta$ , ma non per  $o$  e  $\omega$

#### Simmetria

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

#### Simmetria trasposta

- ›  $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
- ›  $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

#### Somma

Se  $f_1(n) = O(g_1(n))$  e  $f_2(n) = O(g_2(n))$  allora

$$f_1(n) + f_2(n) = O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$$

#### Prodotto

Se  $f_1(n) = O(g_1(n))$  e  $f_2(n) = O(g_2(n))$  allora

$$f_1(n) \cdot f_2(n) = O(g_1(n)) \cdot O(g_2(n)) = O(g_1(n) \cdot g_2(n))$$

#### Moltiplicazione per una costante

Se  $f(n) = O(g(n))$  e  $a > 0$  allora

$$a \cdot f(n) = O(g(n))$$

### 2.1.8 In equazioni

? Come interpretare la formula

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

Il polinomio  $2n^2 + 3n + 1$  è composto dalla somma di un termine quadratico ( $2n^2$ ) con un termine asintoticamente lineare ( $3n + 1$ )

Tutto ciò vale anche per  $O$ ,  $o$ ,  $\Omega$ ,  $\omega$

? Come interpretare la formula

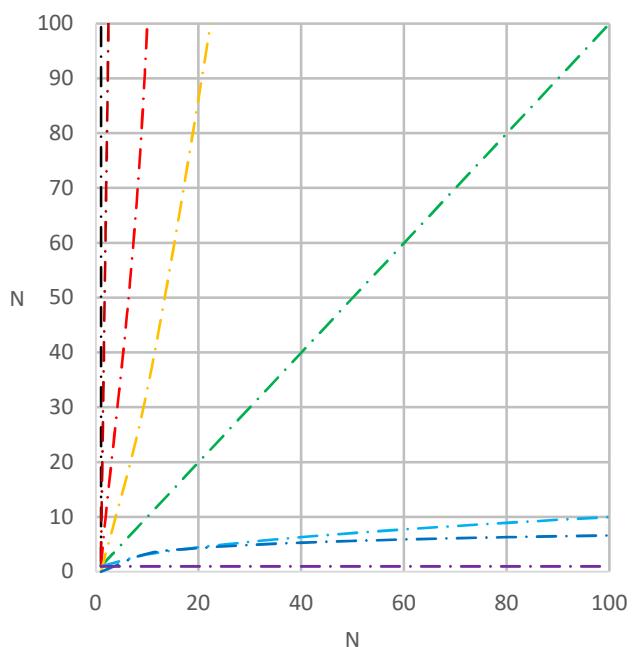
$$2n^2 + \Theta(n) = \Theta(n^2)$$

Esiste qualche funzione  $f(n)$  in  $\Theta(n)$  tale che

$$2n^2 + f(n) = \Theta(n^2)$$

## 2.2 ORDINI DI CRESCITA

| Ordine di crescita | Nome                       |
|--------------------|----------------------------|
| $O(1)$             | Costante                   |
| $O(\log n)$        | Logaritmico                |
| $O(\log^k n)$      | Polilogaritmico $k \geq 1$ |
| $O(n^k)$           | Sublineare $0 < k < 1$     |
| $O(n)$             | Lineare                    |
| $O(n \log n)$      | Pseudolineare              |
| $O(n^k)$           | Polinomiale $k > 1$        |
| $O(n^2)$           | Quadratico                 |
| $O(n^3)$           | Cubico                     |
| $O(c^n)$           | Esponenziale, base $c > 1$ |
| $O(n!)$            | Fattoriale                 |
| $O(n^n)$           | Esponenziale, base $n$     |



## 2.3 COMPLESSITÀ COMPUTAZIONALE

### Complessità computazionale di un algoritmo

Un algoritmo  $\mathcal{A}$  ha complessità computazionale  $O(f(n))$  rispetto a una certa risorsa di calcolo se la quantità di risorse necessaria per eseguire  $\mathcal{A}$  su un qualsiasi input di dimensione  $n$  corrisponde a  $O(f(n))$ .

### Complessità computazionale di un problema

Un problema  $\mathcal{P}$  ha complessità computazionale  $O(f(n))$  rispetto a una certa risorsa di calcolo se il migliore algoritmo che risolve  $\mathcal{P}$  ha costo computazionale  $O(f(n))$  rispetto a tale risorsa.

- ! non c'è differenza tra "costo" e "complessità" computazionale
- ! Le **risorse di calcolo** considerate di solito sono:
  - > **Tempo di esecuzione**
  - > **Occupazione di memoria**
- ! Le definizioni valgono anche per le altre notazioni asintotiche

### 2.3.1 Analisi del caso ottimo, pessimo e medio

Quando parliamo di complessità computazionale degli algoritmi ci capiterà di dover esprimere quale sia *almeno, al peggio o in media* la complessità computazionale di un algoritmo.

- > Caso ottimo: comportamento in **condizioni ottimali**
  - o Ricerca in un array quando l'elemento cercato è il primo
- > Caso pessimo: comportamento in **condizioni sfavorevoli**
  - o Ricerca in un array quando l'elemento è l'ultimo o assente
- > Caso medio: comportamento **medio**, su tutti i possibili input

Esempio: Ricerca lineare

- Cercare la posizione di un valore all'interno di un array ( se non trovato)

```
1. function linSearch(array A[1···n], int x) → int
2.   for i=1, ..., n do
3.     if A[i]==x then
4.       return i
5.   return -1
```

- # Caso ottimo: (x è il primo elemento)  $O(1)$
- # Caso pessimo: (x non presente o ultimo)  $\Theta(n)$

### # Caso medio

- Probabilità uniforme:  $x$  in posizione  $i$  con probabilità  $P_i = 1/n$
- Questa assunzione (che la probabilità sia uniforme) non è sempre vera
- Senza altre info è il massimo che possiamo fare

Dobbiamo però considerare anche se l'elemento non viene trovato

- Probabilità uniforme:  $x$  in posizione  $i$  con probabilità  $P_i = \frac{1}{n+1}$ 
  - $n+1$  per simulare l'eventuale assenza dell'elemento nell'array di dimensione  $n$
- Il tempo necessario per ispezionare la posizione  $i$  è  $T_i = i$  passi
  - Per arrivare a  $x$  in posizione  $i$  eseguiamo sequenzialmente  $i$  passi
  - Quindi cercare un elemento non presente costa  $T_{n+1} = n+1$
- Per calcolare il costo medio sommiamo le probabilità di ispezione ( $P_i$ ) moltiplicate per il rispettivo tempo di ispezione ( $T_i$ )

$$\text{Costo medio} = \sum_{i=1}^{n+1} P_i T_i = \frac{1}{n+1} \sum_{i=1}^{n+1} i = \frac{1}{n+1} \cdot \frac{(n+1)(n+2)}{2} = \frac{n+2}{2} = \Theta(n)$$

Esempio: Ricerca binaria

- Cercare la posizione di un valore in un array ordinato (-1 se non trovato)

```
1. Function binSrc(array A[1···n], int x) → int
2.   i=1, j=n
3.   While i≤j do
4.     m=(i+j)/2
5.     if A[m]==x then
6.       return m
7.     else if A[m]<x then
8.       i=m+1
9.     else
10.      j=m-1
11.   return -1
```

| Iterazione  | 1   | 2     | 3     | ... | $i$         |
|-------------|-----|-------|-------|-----|-------------|
| Dim. Spazio | $n$ | $n/2$ | $n/4$ | ... | $n/2^{i-1}$ |

- # Caso ottimo ( $x$  nella posizione centrale  $(1+n)/2$ ):  $O(1)$

- # Caso pessimo ( $x$  non presente nell'array):

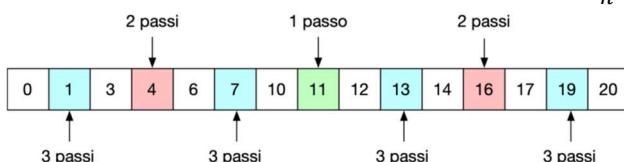
❓ Qual è il massimo numero di iterazioni del ciclo while?  
Dopo ogni iterazione lo spazio di ricerca viene dimezzato

Il ciclo while termina quando la dimensione dello spazio è  $< 1$ ; Il numero max di iterazioni si ottiene:

$$\begin{aligned} n/2^{i-1} &\Rightarrow n < 2^{i-1} \\ &\Rightarrow \log_2 n < \log_2 2^{i-1} \\ &\Rightarrow \log_2 n < (i-1) \cdot \log_2 2 \\ &\Rightarrow i > \log_2(n) + 1 \end{aligned}$$

# Caso medio:

probabilità uniforme:  $x$  in posizione  $i$ ,  $P_i = \frac{1}{n}$



$$\frac{1 \times 1 + \dots + i \times 2^{i-1} + \dots + \log_2 n \times 2^{\log_2 n-1}}{n} = \frac{1}{n} \sum_{i=1}^{\log_2 n} i \cdot 2^{i-1} \leq \frac{\log_2 n}{n} \sum_{i=2}^{\log_2 n} 2^i = \frac{\log_2 n}{n} \cdot \frac{2^{\log_2 n+1} - 2}{2-1} = O(\log n)$$

Esempio: Valore minimo

➤ Cercare la posizione del valore minimo in un array

```

1. Function min(array A[1...n]) → int
2.   m=1
3.   for i=2,..., n do
4.     if (A,i.<A[m] then
5.       m=i
6.   return m

```

- # Il ciclo for viene eseguito *sempre* esattamente  $n - 1$  volte
  - Le istruzioni eseguite in ogni iterazione hanno costo fisso  $O(1)$
- # Casi ottimo, pessimo e medio coincidono:  $O(n)$ 
  - ⇒  $\Theta(n)$  più preciso perché l'algoritmo viene eseguito  $n$  volte, indipendentemente dall'array

### 2.3.2 Analisi ammortizzata

L'**analisi ammortizzata** è uno strumento utile a valutare il costo medio di una sequenza di operazioni

Per alcuni algoritmi una data operazione può essere molto costosa o molto efficiente a seconda delle situazioni quindi non ci basta calcolare solamente il costo medio se lavoriamo su più operazioni

Abbiamo due metodi principali per l'analisi ammortizzata:

#### 2.3.2.1 Metodo dell'aggregazione

Determiniamo un *upper bound* al costo totale di una sequenza di  $n$  operazioni e poi dividiamo per  $n$

#### 2.3.2.2 Metodo degli accantonamenti

1. Assegniamo un costo ammortizzato a ogni operazione
2. Il costo di ogni operazione viene addebitato dal costo ammortizzato
  - Se il costo ammortizzato è maggiore di quello effettivo si guadagna credito per "fare cassa"
3. Continuiamo a "fare cassa" finché non incontreremo un costo effettivo maggiore di quello ammortizzato
  - Li potremo usare i nostri risparmi per coprire la spesa maggiore
4. Il costo ammortizzato è corretto se il credito non è mai negativo.

Esempio: contatore binario

- Operazione di incremento di un numero binario su di un array
- La cifra più significativa è nella prima posizione dell'array

```

1. function incr[Array A[1...k]
2.   i = k
3.   while i ≥ 1 && A[i] == 1 do
4.     A[i] = 0
5.     i = i-1
6.   if i≥1 then
7.     A[i] = 1

```

- # Caso ottimo: ( $A[k]$  contiene zero):  $O(1)$
- # Caso pessimo: ( $A$  contiene tutti uno):  $O(k)$

#### - Aggregazione

Sommiamo i costi per i cambi di bit:

- Il  $k$ -esimo bit è cambiato a ogni incremento

$$\begin{aligned}
 n + n/2 + \dots + n/2^{k-1} &= \sum_{i=0}^{k-1} \frac{n}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = \\
 &= n \frac{1}{1 - 1/2} = 2n \rightarrow O(n) \\
 \Rightarrow \frac{O(n)}{n} &= O(1)
 \end{aligned}$$

#### - Accantonamenti

Addebitiamo un costo ammortizzato di 2€ ( $0 \rightarrow 1$ )

- Usiamo 1€ per pagare lo switch

- Salviamo l'altro 1€ in credito

- Il credito servirà per il prossimo  $1 \rightarrow 0$

! Ogni 1 ha 1€ di credito, in ogni momento

! Una sequenza di  $n$  incrementi costa  $2n$ €

Quindi il costo ammortizzato di ogni incremento è pari a  $\frac{2n}{n} = O(1)$

## 2.4 EQUAZIONI DI RICORRENZA

Una **equazione di ricorrenza** descrive ogni elemento in una sequenza in termini degli elementi precedenti

Abbiamo già visto l'equazione di ricorrenza di Fibonacci

$$T(n) = \begin{cases} 1 & n \leq 2 \\ T(n-1) + T(n-2) + 1 & n > 2 \end{cases}$$

Vedremo due metodi per risolvere le equazioni di ricorrenza:

- Metodo dell'iterazione
- Master Theorem

### Notazione:

Usciamo 1 al posto di costanti simboliche e sostituiamo ove possibile la notazione asintotica con espressioni positive, quindi

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\lfloor n/3 \rfloor) + O(n) & n > 1 \end{cases} \quad \xrightarrow{\text{diventa}} \quad T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{3}) + n & n > 1 \end{cases}$$

### 2.4.1 Metodo dell'iterazione

Sostituendo la parte ricorsiva nell'equazione, riconducendoci a uno schema ricorsivo.

Esempio: consideriamo la ricorrenza  $T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &= T(n/8) + c + c + c \\ &= \dots \\ &= T(n/2^i) + c \cdot i \end{aligned}$$

La ricorsione termina quando giungiamo al caso base, quindi quando:

$$n/2^i = 1 \rightarrow 2^i = n \rightarrow i = \log_2 n$$

Per calcolare  $T(n)$  sommiamo anche l'iterazione del caso base:

$$T(n) = T(1) + c \cdot \log_2 n = \Theta(\log n)$$

Esempio: Metodo dell'iterazione

Consideriamo la ricorrenza  $T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$

$$\begin{aligned} T(n) &= T(n/2) + n \\ &= T(n/4) + n/2 + n \\ &= T(n/8) + n/4 + n/2 + n \\ &= \dots \\ &= T(n/2^i) + n \sum_{k=0}^{i-1} 1/2^k \\ &= T(n/2^i) + n \frac{1/2^i - 1}{1/2 - 1} \\ &= T(n/2^i) + 2n(1 - 1/2^i) \end{aligned}$$

La ricorsione termina quando  $n/2^i = 1$  ovvero quando  $i = \log_2 n$ ; quindi

$$\begin{aligned} T(n) &= T(1) + 2n \left(1 - \frac{1}{n}\right) \\ &= 1 + 2n - 2 = 2n - 1 \\ &= \Theta(n) \end{aligned}$$

### 2.4.2 Master Theorem

Il **Master theorem** è un approccio per risolvere ricorrenze della forma:

$$T(n) = aT(n/b) + f(n)$$

Con  $a \geq 1$  e  $b > 1$  costanti e  $f(n)$  asintoticamente positiva.

#### Master Theorem:

Si consideri la seguente equazione di ricorrenza

$$T(n) = \begin{cases} d & n = 1 \\ aT(n/b) + cn^\beta & n > 1 \end{cases}$$

Dove  $a \geq 1, b > 1$  e  $c, d$  costanti.

Sia  $\alpha = \log_b a = \frac{\log a}{\log b}$ . Allora

1. Se  $\alpha > \beta \Rightarrow T(n) = \Theta(n^\alpha)$
2. Se  $\alpha = \beta \Rightarrow T(n) = \Theta(n^\alpha \log n)$
3. Se  $\alpha < \beta \Rightarrow T(n) = \Theta(n^\beta)$

Esempi: Risolvere col Master Theorem:

1.  $T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$

Abbiamo:

$$\begin{aligned} a &= 1 \\ b &= 2 \rightarrow \alpha = \log_b a = \log_2 1 = 0, \quad \beta = 0 \\ \alpha &= \beta \rightarrow, \quad T(n) = \Theta(n^\alpha \log n) = \Theta(\log n) \end{aligned}$$

2.  $T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$

Abbiamo:

$$\begin{aligned} a &= 1 \\ b &= 2 \rightarrow \alpha = \log_b a = \log_2 1 = 0, \quad \beta = 1 \\ \alpha &< \beta \rightarrow, \quad T(n) = \Theta(n^\beta) = \Theta(n) \end{aligned}$$

3.  $T(n) = \begin{cases} 1 & n = 1 \\ 3T(n/4) + n^2 & n > 1 \end{cases}$

Abbiamo:

$$\begin{aligned} a &= 3 \\ b &= 4 \rightarrow \alpha = \log_b a = \log_4 3 \approx 0.79, \quad \beta = 2 \\ \alpha &< \beta \rightarrow, \quad T(n) = \Theta(n^\beta) = \Theta(n^2) \end{aligned}$$

### 3 ALGORITMI DI ORDINAMENTO

Nota: non è necessario memorizzare lo pseudocodice dei vari algoritmi, focalizzarsi sul funzionamento e sul costo computazionale

#### 3.1 INTRODUZIONE

Come funziona un algoritmo di ordinamento?

- **Input:** una sequenza di  $n$  numeri  $[a_1, a_2, \dots, a_n]$
- **Output:** una permutazione  $p: \{1, \dots, n\} \rightarrow \{1, \dots, n\}$  degli indici

Esempio:

|                             |                              |
|-----------------------------|------------------------------|
| IN: [7, 32, 88, 21, 92, -4] | OUT: [-4, 7, 21, 32, 88, 92] |
|-----------------------------|------------------------------|

##### 3.1.1.1 Tre categorie di algoritmi di ordinamento

- # Incrementali: SelectionSort, InsertionSort
- # Divide et Impera: MergeSort, QuickSort
- # Non-comparativi: CountingSort, RadixSort

##### 3.1.1.2 Chiave e valore

Assumendo di avere in input un array, ogni elemento dell'array ha:

- Chiave: → è il criterio usato per ordinare gli elementi
- Valore: → è il contenuto associato alla chiave

Es. se stiamo ordinando un array di persone *per età*,

- |        |                         |
|--------|-------------------------|
| chiave | → l'età di ogni persona |
| valore | → il nome della persona |

La distinzione tra chiave e valore è importante, perché l'algoritmo di ordinamento deve essere in grado di ordinare in base alla chiave mantenendo allo stesso tempo il valore associato ad ognuna delle chiavi.

##### 3.1.1.3 Proprietà degli algoritmi di ordinamento

L'algoritmo inoltre può essere:

- in place: → l'algoritmo riordina nell'array in input, non è richiesto nessun array aggiuntivo
- stabile: → valori con la stessa chiave escono in output nello stesso ordine con cui sono entrati in dell'input.

### 3.2 LA STRUTTURA DATI ARRAY

Un array è una struttura dati costituita da una sequenza di elementi omogenei accessibili tramite un indice.

È possibile accedere agli elementi tramite il loro indice

#  $A[1]=4, A[5]=3$

# Il costo di accesso è *costante* per tutti gli elementi

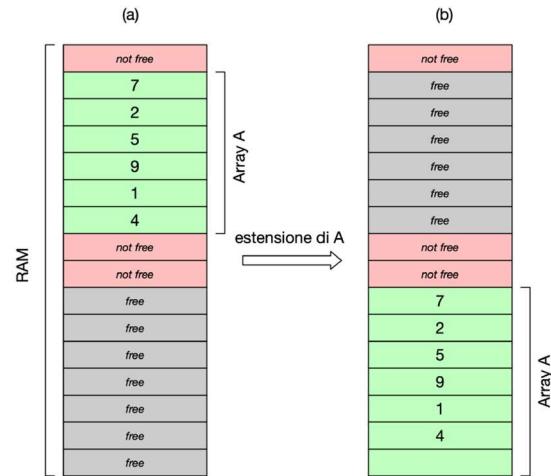
|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 7 | 2 | 5 | 9 | 1 | 4 |

Gli array sono una struttura dati **a dimensione fissa**.

- La dimensione è fissata quando l'array è creato
- Un array può essere ridimensionato (ampliato) solo creando un nuovo array più grande e copiando gli elementi del vecchio array nel nuovo.

### Ridimensionamento di un array:

- Non c'è spazio contiguo in RAM per estendere A
- Estendiamo A spostandolo in una nuova posizione della RAM



### 3.3 INCREMENTALI

Algoritmi semplici che costruiscono l'ordinamento un elemento per volta.

se partendo da un prefisso ordinato  $A[1, \dots, k]$  dell'intero array (da  $k$  a  $n$  non sono più ordinati) estendiamo l'ordinamento a un prefisso più ampio  $A[1, \dots, k + 1]$

- # SelectionSort  
Cerca il minimo in  $A[k + 1, \dots, n]$  e lo sposta in posizione  $k + 1$
- # InsertionSort  
Inserisce l'elemento  $A[k + 1]$  nella corretta posizione in  $A[1, \dots, k + 1]$

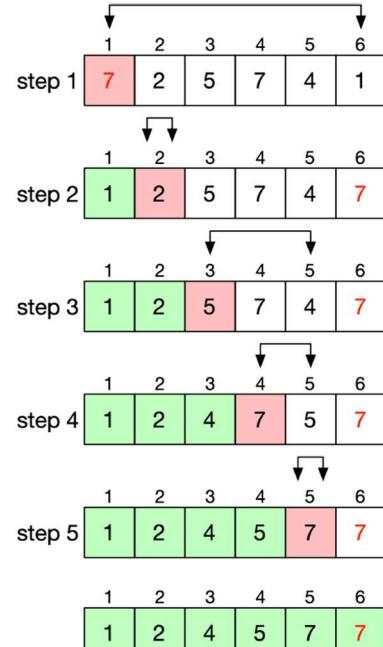
#### 3.3.1 SelectionSort $\rightarrow \Theta(n^2)$

A ogni passo  $i = 1, \dots, n - 1$

- cerca la posizione  $j$  della minima chiave in  $A[i, \dots, n]$
- scambia (swap)  $A[j]$  con  $A[i]$

```

1. function selectionSort (A[1,...,n])
2.   for i=1,...,n-1 do
3.     // cerca min in A[1,..., n]
4.     m=i
5.     for j=i+1,...,n do
6.       if A[j]<A[m] then
7.         m=j
8.       // swap A[m] with A[i]
9.       if m≠i then
10.         swap (A, i,m)
11
12. function swap (A[1,...,n], int i, int j)
13.   tmp = A[i]
14.   A[i] = A[j]
15.   A[j] = tmp
  
```



La funzione swap scambia due elementi nell'array

Lo swap a riga 10. Rende l'algoritmo non stabile

Il ciclo a riga 2. Viene eseguito  $n - 1$  volte

I due cicli vengono eseguiti interamente ogni volta  $\Rightarrow$  caso ottimo, medio e pessimo coincidono

|                              |                 |  |
|------------------------------|-----------------|--|
| Ricerca del minimo (r: 4-7): | $\Theta(n - i)$ | $(n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^{n-1} (n - i) = \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2} = \Theta(n^2)$ |
| Swap (r: 12-15):             | $O(1)$          |  |

### 3.3.2 InsertionSort $\rightarrow \Theta(n^2)$

A ogni passo  $i = 2, \dots, n$

- ›  $A[1, \dots, i-1]$  è ordinato
- › inserisci  $A[i]$  nella posizione corretta in  $A[1, \dots, i]$

è l'approccio intuitivo che utilizziamo per ordinare le carte da gioco

```

1. function 18iglio18onSort(A[1,...,n])
2.   for i = 2, ..., n do
3.     j = i
4.     while j > 1 && A[j] < A[j-1] do
5.       swap(A, j, j-1)
6.       j = j - 1

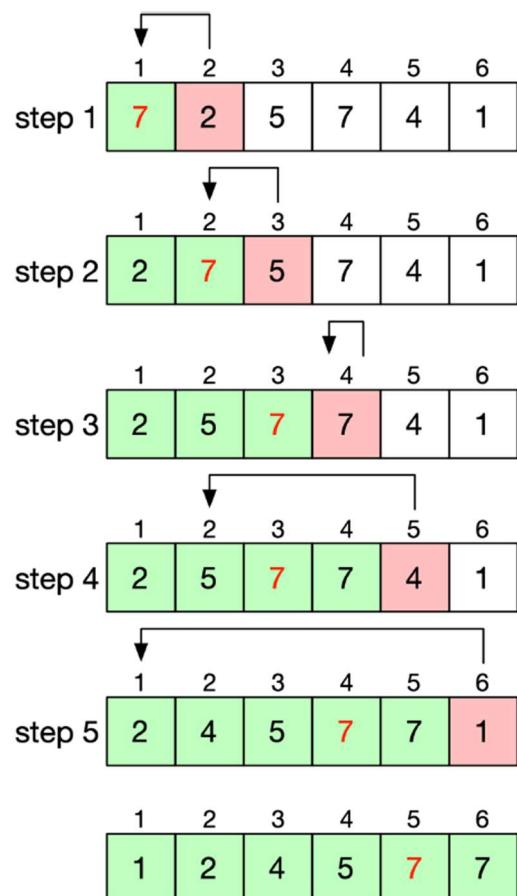
```

Il ciclo for a riga 2. Viene eseguito sempre interamente  
Il ciclo while a riga 4 potrebbe non essere eseguito

#### Costo caso pessimo (array ordinato al contrario)

- › Le chiavi ordinate dalla più grande alla più piccola
- › Il ciclo for viene eseguito  $n-1$  volte
- ›  $\forall i$ , il ciclo while viene eseguito  $i-1$  volte

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



#### Costo caso ottimo (array già ordinato)

- › L'array già ordinato
- › Il ciclo for viene eseguito  $n-1$  volte
- › Il ciclo while non viene mai eseguito

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

Nota: anche per array quasi ordinati

$$\Theta(n)(\text{ciclo for}) + O(nk)(\text{ciclo while}) = O(nk)$$

Se  $k$  è costante rispetto a  $n$  il ciclo while ha costo  $O(n)$ , il costo complessivo rimane invariato

#### Costo caso medio

- › Il ciclo for viene sempre eseguito interamente
- › Ad ogni iterazione  $i$  del ciclo for, il ciclo while viene eseguito al max  $i-1$  volte
- ⇒ il ciclo while viene eseguito in media  $(i-1)/2$  volte

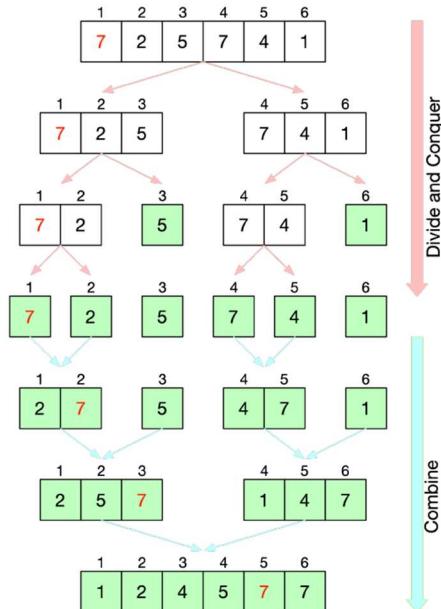
$$\sum_{i=2}^n \frac{i-1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{1}{2} \frac{n(n-1)}{2} = \Theta(n^2)$$

### 3.4 DIVIDE ET IMPERA

Detti anche “divide and conquer” ma cosa intende?

- # Divide: → dividere il problema in sottoproblemi più piccoli
- # Conquer: → risolvere (“conquistare”) i sotto-problemi ricorsivamente
- # Combine: → fondere le soluzioni ai sotto-problemi in una più generale

#### 3.4.1 MergeSort → $\Theta(n \log n)$



- **Divide:** dividiamo  $A[1, \dots, n]$  in due metà  $A_1 = A[1, \dots, q]$  e  $A_2 = [q + 1, \dots, n]$
- **Conquer:** richiamiamo ricorsivamente l’algoritmo su  $A_1, A_2$  se hanno lunghezza  $> 1$
- **Combine:** combiniamo i due array ordinati  $A_1, A_2$  in un unico array ordinato

```

1. function mergeSort(A[1,...,n], p, r)
2.   if p < r then
3.     q = (p+r)/2.
4.     mergeSort(A, p, q)
5.     mergeSort(A, q+1, r)
6.     merge(A, p, q, r)

```

```

1. function merge(A[1,...,n], p, q, r)
2.   let B=[1,...,(r-p+1)]
3.   i = p
4.   j = q+1
5.   k = 1
6.   while i <= q && j <= r do
7.     if A[i] <= A[j] then
8.       B[k] = A[i]
9.       i++
10.    else
11.      B[k] = A[j]
12.      j++
13.      k++
14.   while i <= q do
15.     B[k] = A[i]
16.     k++, i++
17.   while j <= r do
18.     B[k] = A[j]
19.     k++, j++
20.   for k=1,...,(r-p+1) do
21.     A[p+k-1] = B[k]

```

Dopo il merge il sotto-array  $A[p, \dots, r]$  è ordinato



Alla fine ricopiamo B in A

La funzione merge combina gli array ordinati  $A[p, \dots, q], A[q + 1, \dots, r]$

L’equazione di ricorrenza è

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + f(n) & n > 1 \end{cases} \xrightarrow{\text{M.T.}} T(n) = \Theta(n \log n)$$

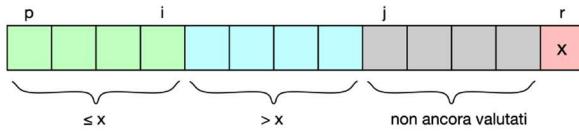
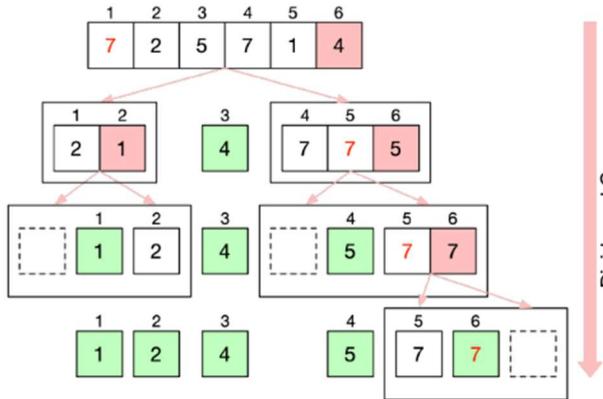
dove  $f(n) = \Theta(n)$  (merge)

### 3.4.2 QuickSort $\rightarrow \Theta(n \log n)$

Si sceglie un elemento dell'array detto "pivot", e si divide l'array in due parti, in modo che gli elementi  $\leq$  al pivot siano a sinistra, mentre i  $>$  sono a destra. Questo processo viene ripetuto per le sottoparti fino a quando ogni sotto-array ha dimensione 1 o 0. Infine gli array ordinati vengono uniti assieme.

```

1. function quickSort(A[1,...,n], p, r)
2.   if p < r then
3.     q = partition(A, p, r)
4.     quickSort(A, p, q-1)
5.     quickSort(A, q+1, r)
    
```



Costo caso ottimo, medio e pessimo di partition

$$\Theta(r - p + 1)$$

```

1. function partition(A[1,..., n], p, r) -> int
2.   x = A[r]
3.   i = p - 1
4.   for j = p, ..., r-1 do
5.     if A[j] <= x then
6.       swap(A, i+1, j)
7.       i++
8.   swap(A, i+1, r)
9.   return(i + 1)
    
```

Il costo computazionale di QuickSort dipende da quanto bene `partition` riesce a partizionare l'array nei due sotto-array

La scelta del pivot comporta sotto-array bilanciati o sbilanciati

- Sono bilanciati se hanno approssimativamente la stessa lunghezza
- Sono sbilanciati se le loro lunghezze differiscono di molto

#### Partizionamento pessimo:

(un array ha lunghezza 0 e l'altro  $n - 1$ )

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

Possiamo risolvere con metodo iterativo:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + n \\ &= T(n-1) + 1 + n \\ &= T(n-2) + 2 + (n-1) + n \\ &= T(n-3) + 3 + (n-2) + (n-1) + n \\ &= \dots \\ &= T(n-i) + i + \sum_{k=0}^{i-1} n-k \end{aligned}$$

La ricorsione termina per  $n - i = 0 \Rightarrow i = n$

$$\begin{aligned} T(n) &= 1 + n + \sum_{k=0}^{n-1} (n-k) = 1 + n + \sum_{k=1}^n k \\ &= 1 + n + \frac{n(n+1)}{2} = \Theta(n^2) \end{aligned}$$

#### Partizionamento ottimo:

(i due sotto-array sono entrambi lunghi  $n/2$ )

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

Possiamo usare il Master Theorem ( $\alpha = \beta = 1$ )

Il costo nel caso ottimo è  $T(n) = \Theta(n \log n)$

#### Caso medio:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(i) + T(n-i-1) + n & n > 1 \end{cases}$$

Ma  $i$  e  $n - i - 1$  possono cambiare a ogni chiamata ricorsiva

Quindi consideriamo tutte le partizioni equiprobabili  $\Rightarrow$

$$T(n) = \begin{cases} 1 & n \leq 1 \\ \frac{1}{n} \sum_{i=0}^{n-1} [T(i) + T(n-i-1)] + n & n > 1 \end{cases}$$

Col metodo di sostituzione otteniamo

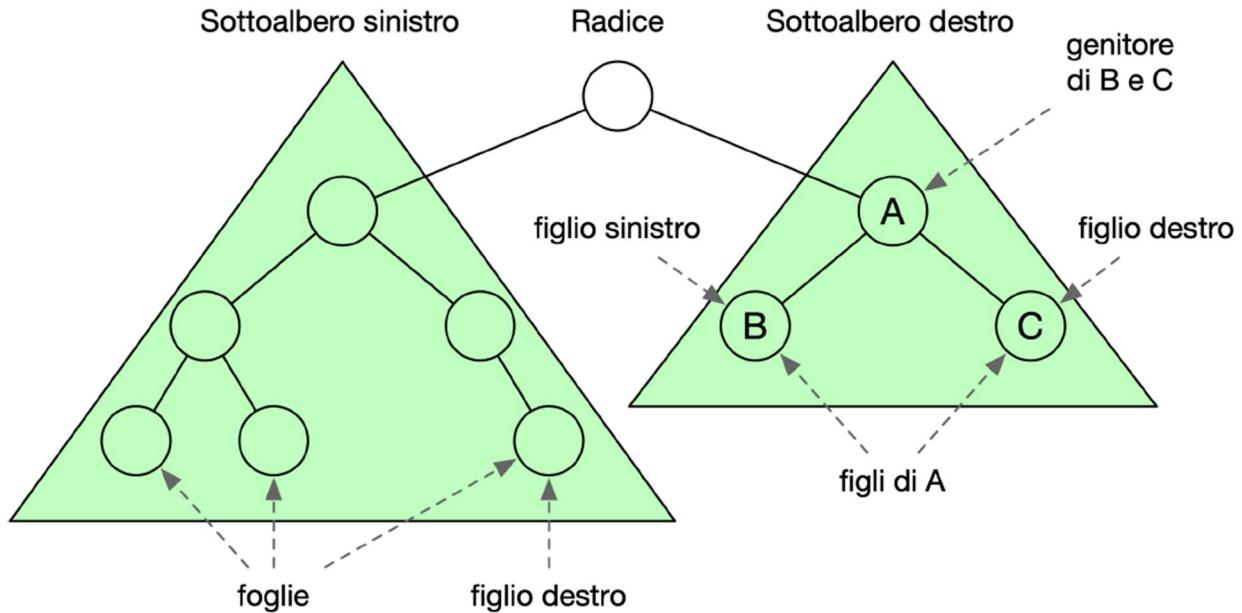
$$T(n) = O(n \log n)$$

### 3.4.3 Riassunto algoritmi comparativi

- ❓ Esistono algoritmi di ordinamento comparativi con prestazioni migliori di  $O(n \log n)$  ?

Rappresentiamo con **alberi di decisione** gli algoritmi comparativi che abbiamo visto.

#### 3.4.3.1 Alberi binari e le loro proprietà



- # La profondità di un nodo è la lunghezza del percorso che va dalla radice al nodo
- # L'altezza di un albero è la sua massima profondità

Se vogliamo rappresentare un algoritmo di ordinamento comparativo con un albero di decisione:

- > Ogni nodo dove corrispondere al confronto di un paio di elementi dell'array
- > **L'altezza dell'albero** corrisponde al numero di confronti nel **caso pessimo**
- > **L'altezza media** corrisponde al numero di confronti nel **caso medio**
- > Il numero di foglie dell'albero è  $\geq n!$

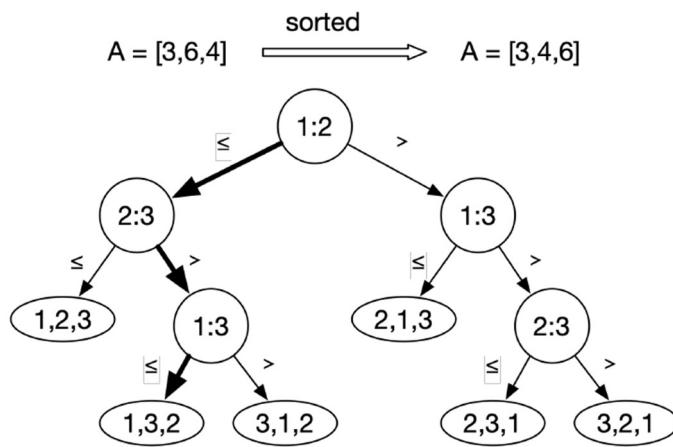


Figura 1: Albero di decisione dell'InsertionSort

NOTA: Il numero di foglie è  $\geq n!$  Perché ogni foglia rappresenta una possibile *permutazione* degli elementi dell'array di input

**Teorema:**

Sia  $T_k$  un albero binario con  $k$  foglie in cui ogni nodo interno ha esattamente due figli.

E sia  $h(T_k)$  l'altezza di  $T_k$ ; Allora:

$$h(T_k) \geq \log_2 k$$

**Teorema:**

Ogni algoritmo di ordinamento comparativo richiede  $\Omega(n \log n)$  confronti nel caso pessimo

### 3.5 NON-COMPARATIVI

Ponendo determinanti vincoli sul tipo di elementi da ordinare, possiamo lavorare con algoritmi che in caso pessimo ordinano in tempo lineare.

Noi vedremo:

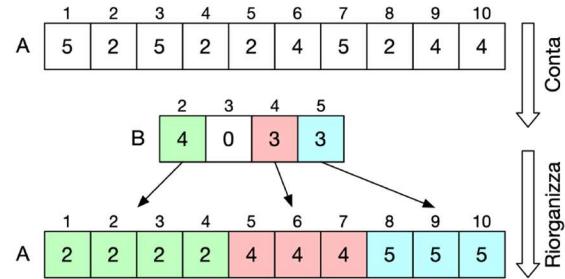
- CountingSort
- RadixSort

#### 3.5.1 CountingSort $\rightarrow \Theta(n + k)$

Assunzioni: le chiavi da ordinare sono interi nell'intervallo  $[a, b]$

L'idea:

- › Usa un array  $B[ ]$  di lunghezza  $b - a + 1$  per *contare* quante volte ogni valore chiave  $x \in [a, b]$  appare nell'array in input  $A[ ]$
- › Riorganizziamo questi valori in  $A[ ]$



2: Si conta quante volte ogni valore chiave appartenente all'intervallo appare nell'array in input e si riorganizzano tali valori in ordine

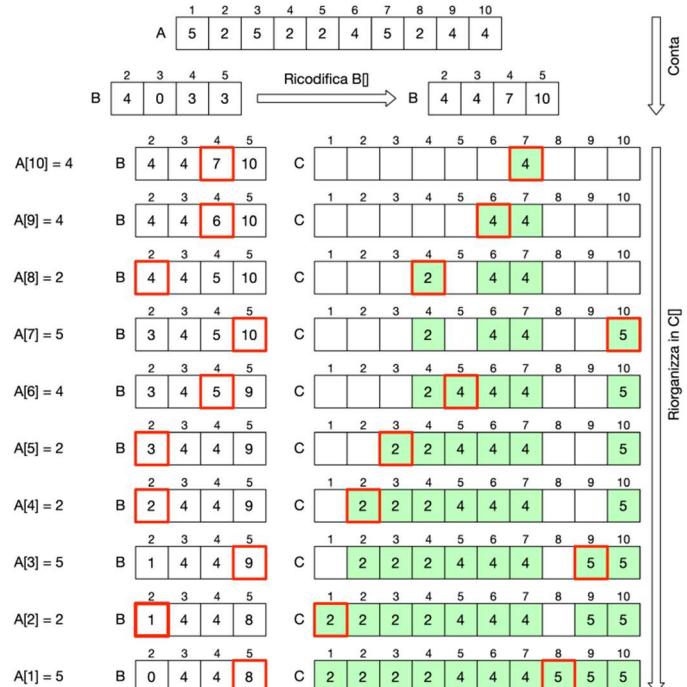
```

1. function countingSort(A[1,..., n])
2.   a = min(A)
3.   b = max(A)
4.   k = b - a + 1
5.   Let B=[1,...,k] be a new array
6.   for i = 1, ..., k
7.     B[i] = 0
8.   for i = 1, ..., n do
9.     B[A[i] - a + 1]++
10.  j = 1
11.  for i = 1, ..., k do
12.    while B[i] > 0 do
13.      A[j] = i + a - 1
14.      B[i]-
15.      j++

```

Costo Ottimo, medio e pessimo:

$$\Theta(n + k) = \Theta(\max(n, k))$$

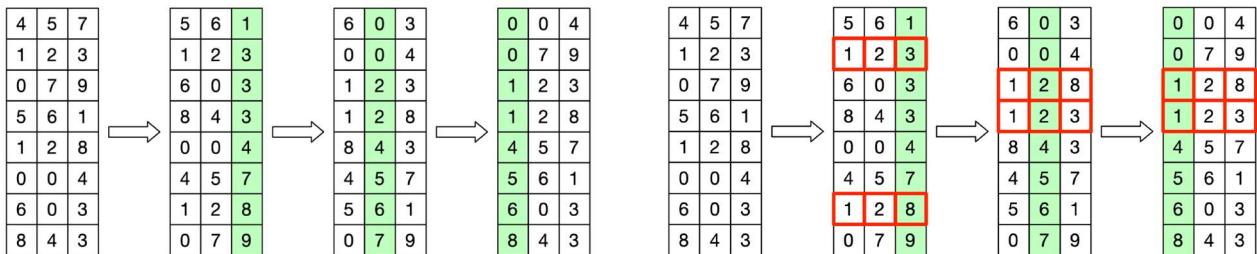


### 3.5.2 RadixSort

Assunzioni: i valori chiave sono composti da cifre o caratteri

L'idea:

- › Ordina prima rispetto alla cifra meno significativa di ogni elemento, poi rispetto alla penultima cifra meno significativa e così via.
- › L'ordinamento delle cifre deve essere stabile



```

1. function radixSort(A[1,...,n])
2.   d = max_key_length
3.   for i = 1,...,d do
4.     ordinamento stabile dell'array A sulla i-esima cifra delle keys

```

Il massimo numero di cifre in una chiave è  $d$

Le cifre sono numerate da 1 a  $d$  da destra verso sinistra

Se usiamo countingSort come algoritmo stabile il costo minimo e pessimo di radixSort è  $\Theta(d(n + k))$  dove:

- $d$  è il numero massimo di cifre in una chiave
- $k$  è il numero di possibili valori per una cifra
- Se le chiavi sono interi  $k = 10$ , mentre se le chiavi sono stringhe  $k = \text{numero di caratteri ammessi}$ .

Se  $k = O(n)$  e  $d$  è un valore costante  $\Rightarrow$  il costo è  $\Theta(n)$  Lineare sul numero di elementi in  $A[]$

## 3.6 RIASSUNTO

|               | Caso ottimo        | Caso medio         | Caso pessimo  |
|---------------|--------------------|--------------------|---------------|
| SelectionSort |                    | $\Theta(n^2)$      |               |
| InsertionSort | $\Theta(n)$        | $\Theta(n^2)$      | $\Theta(n^2)$ |
| MergeSort     |                    | $\Theta(n \log n)$ |               |
| QuickSort     | $\Theta(n \log n)$ | $O(n \log n)$      | $\Theta(n^2)$ |
| CountingSort  |                    | $\Theta(n + k)$    |               |
| RadixSort     |                    | $\Theta(d(n + k))$ |               |

|               | In place | Stabile |
|---------------|----------|---------|
| SelectionSort | Si       | No      |
| InsertionSort | Si       | Si      |
| MergeSort     | No       | Si      |
| QuickSort     | Si       | No      |

## 4 STRUTTURE DATI ELEMENTARI

I dati sono organizzati in memoria tramite *strutture dati* in modo da organizzare i dati e offrire una serie di operazioni per manipolare la struttura.

Esistono diversi metodi di definire strutture dati e le loro interfacce pubbliche, come ad esempio le interfacce Java o i template in C++.

### 4.1.1 Esempio: Dizionari

```
1. public interface Dizionario {
2.     public void insert (Comparable key, Object value);
3.     public void delete (Comparable key);
4.     public Object search (Comparable key);
5. }
```

Decidiamo per esempio di inserire gli elementi come chiavi ( $k, v$ ) in un array dinamico **ordinato**.

I nostri metodi faranno dunque le seguenti operazioni:

- # **search:** ricerca binaria sull'array  $O(\log_2 n)$
- # **insert:** inserimento ordinato, quindi scorrendo l'array e trovando il punto giusto dove inserire e spostare tutti gli elementi successivi per fare spazio per il nuovo elemento  $O(n)$
- # **delete:** ricerca binaria dell'elemento e rimozione spostando a sinistra tutti gli elementi successivi  $O(n)$

Decidiamo per esempio di inserire gli elementi come chiavi ( $k, v$ ) in una lista.

I nostri metodi faranno dunque le seguenti operazioni:

- > **search:** ricerca sulla lista  $O(n)$
- > **insert:** inserimento a testa  $O(1)$
- > **delete:** ricerca dell'elemento e cambio del puntatore  $O(n)$

come si può notare varie implementazioni hanno vari tradeoff.

### 4.1.2 Tipi di strutture dati

- **Linearî:** Elementi in una sequenza dove ogni elemento ha un index (array)
- **Non linearî:** Elementi inseriti in una sequenza senza una linearità in memoria (liste)
- > **Statiche:** Numero di elementi sempre costante nel tempo
- > **Dinamiche:** Numero di elementi che cambia nel tempo
- > **Omogenee:** Dati tutti dello stesso tipo
- > **Non omogenee:** Dati possono essere di tipo diverso

## 4.2 STRUTTURE DATI ELEMENTARI

### 4.2.1 Strutture concatenate (collegate)

Abbiamo due tecniche per rappresentare collezioni di elementi: **strutture collegate (liste)** oppure **vettori indicizzati**.

Analizzeremo in particolare le strutture collegate che sono più interessanti come implementazione:  
Tutte le strutture collegate (liste) hanno la caratteristica comune di essere una collezione di **record**:

- Tutti i record sono collegati tra di loro
- I record sono numerabili partendo dalla testa o dalla coda
- Ogni record contiene un dato

#### 4.2.1.1 Liste bi-linkate

Per alcuni algoritmi può essere svantaggioso avere un riferimento *solo* al nodo successivo e si usano dunque liste bilinkate dove ogni record ha un puntatore all'elemento precedente e a quello successivo.  
Tuttavia, l'aggiunta di un puntatore occuperà una word in più in memoria per ogni record. Si ha quindi un minimo overhead sulla memoria.

#### 4.2.1.2 Liste circolari

Possiamo oltretutto estendere le liste bilinkate in modo che il nodo in testa alla lista punti alla coda come suo precedente, e che il nodo in coda punti alla testa come suo successivo.

In questo modo si ha una lista bilinkata circolare che rende molto più veloci operazioni come rimozione in coda o aggiunta in coda.

Chiaramente con questo tipo di liste si può avere un problema per capire quando si è in testa o in coda alla lista.

#### 4.2.1.3 Liste circolari con nodo sentinella

Si possono implementare liste circolari come sopra ma inserendo in testa **un nodo vuoto** il cui compito è quello di segnalare quando si è giunti alla testa della lista.

### 4.2.2 Stack (o pile)

Le pile sono una struttura dati molto diffusa e può essere pensata come una lista dove si aggiunge sempre in testa e si possono rimuovere elementi solo in testa.

In questo modo abbiamo un flusso di dati LIFO (Last In, First Out).

Alcuni metodi previsti per una pila potrebbero essere:

- |                   |   |
|-------------------|---|
| # <b>push:</b>    | aggiunge un elemento in cima alla lista                     |
| # <b>pop:</b>     | rimuove e restituisce l'elemento in cima alla lista         |
| # <b>top:</b>     | restituisce (senza rimuovere) il primo elemento sulla lista |
| # <b>isEmpty:</b> | restituisce un booleano che indica se la lista è vuota      |

Alcuni linguaggi come Java, Bytecode o HackVM funzionano con una interfaccia a pile.

Ci sono due principali tecniche di implementazione:

- > **liste concatenate:** una lista come descritta sopra dove si accede solo all'elemento in testa per i push/pop. [Pro: grandeza infinita](#) [Con: più dispendiosa come memoria](#)
- > **array:** utilizziamo un array statico di dimensione fissa tenendo traccia dell'index dell'ultimo elemento aggiunto (cima della pila)  
[Pro: meno dispendioso a livello di memoria](#) [Con: grandeza fissa limitata](#)

#### 4.2.3 Coda (o queue)

La struttura dati coda, a differenza della pila, segue una politica FIFO (First In, First Out)

Le operazioni standard sono:

- # **enqueue:** aggiunge un elemento in coda
- # **dequeue:** rimuove il primo elemento inserito nella coda
- # **first:** restituisce il primo elemento della coda
- # **isEmpty:** restituisce un booleano che indica se la lista è vuota

Possibili implementazioni sono:

- > **liste:** liste normali o bilinkate, che hanno una performance migliore ma sono decisamente più complicate dal punto di vista implementativo.  
Serve tenere traccia della `head` e `tail` rispettivamente per l'estrazione e l'inserimento.
- > **array circolari:** dimensione limitata, hoverhead minore.

## 5 ALBERI

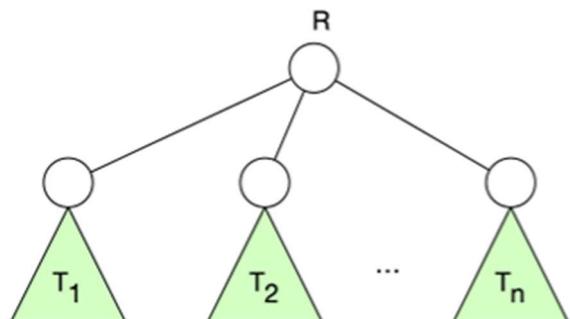
### 5.1.1 Introduzione

Finora abbiamo visto strutture dati sequenziali. Ad un elemento seguiva un altro.

Gli alberi ci consentono di strutturare i dati in modo gerarchico

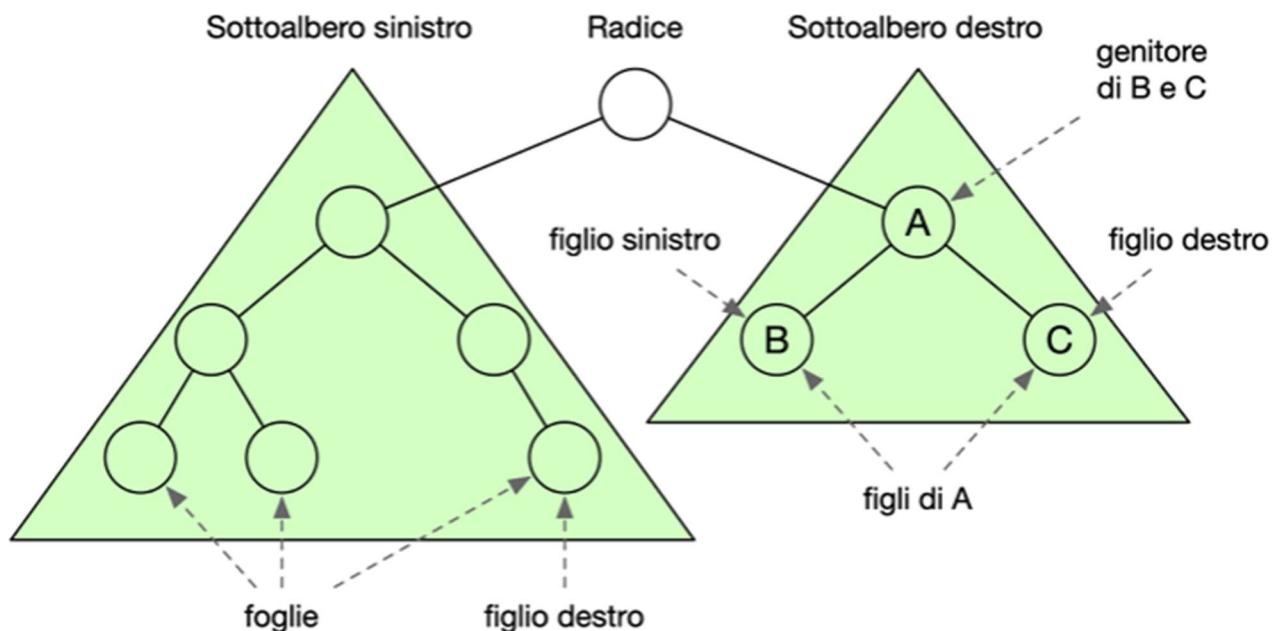
Definizione ricorsiva di **Albero radicato**:

- Insieme vuoto di nodi, oppure
- Una radice  $R$  e zero o più alberi disgiunti (detti *sottoalberi*), dove la radice  $R$  è collegata alla radice di ogni sottoalbero.

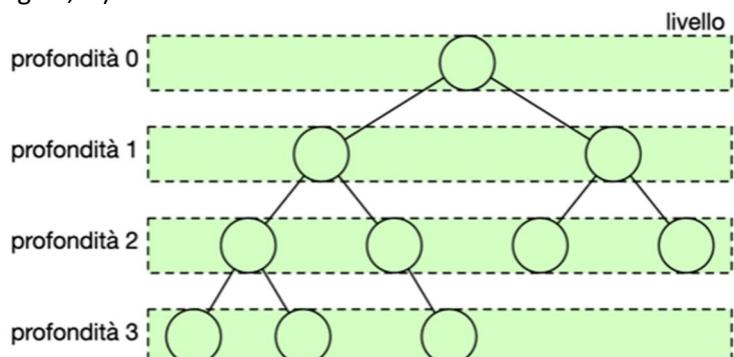


I tipi di alberi più standard sono gli **alberi binari** che contengono 0, 1 o 2 (da cui deriva binario) sottoalberi. Ecco alcune terminologie:

- I nodi che non hanno nessun figlio sono chiamati **foglie**.
- L'albero soprastante al nodo considerato viene definito **genitore**.
- Negli alberi binari, avendo sempre al massimo 2 sottoalberi, possiamo riferirci ad essi come **left node** e **right node**.



- **Profondità**: è il numero di archi attraversati per andare dalla radice al nodo o viceversa (dunque la radice ha profondità = 0, i suoi figli 1, ...)
- **Livello**: è l'insieme di nodi alla stessa profondità.
- **Altezza**: Massima profondità dell'albero



### 5.1.2 Visita

Per le strutture sequenziali (come le liste o gli array) le funzioni di visita sono intuitiva: si può partire dalla fine o dall'inizio e procedere nella direzione scelta.

Per gli alberi abbiamo invece svariate tecniche per visitare tutti i nodi dell'albero:

#### # depth-first search (DFS, in profondità):

Vengono visitati tutti i sottonodi uno dopo l'altro (si visita la prima foglia in fondo, poi la seconda in fondo, etc. ...). Si può svolgere in tre modi diversi:

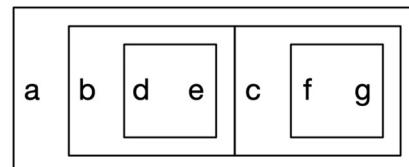
- **pre-ordine:** Visitiamo prima la radice (il nodo passato alla funzione) e svolgiamo una chiamata ricorsiva sul sottoalbero sinistro e una su quello destro.

```

1. function preOrder(Node T)
2.   if T ≠ NIL then
3.     visit(T)
4.     preOrder(T.left)
5.     preOrder(T.right)

```

Caso (ottimo, pessimo, medio):  $\Theta(n)$



- **post-ordine:** Simile a quella pre-ordine ma il nodo radice viene lasciato per ultimo.

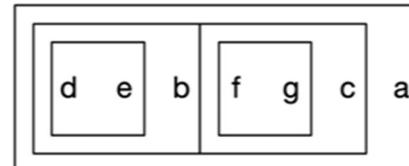
Si fa dunque la chiamata ricorsiva sul nodo di sinistra, poi su quello di destra ed infine si visita la radice.

```

1. function postOrder(Node T)
2.   if T ≠ NIL then
3.     postOrder(T.left)
4.     postOrder(T.right)
5.     visit(T)

```

Costo (ottimo, pessimo, medio):  $\Theta(n)$



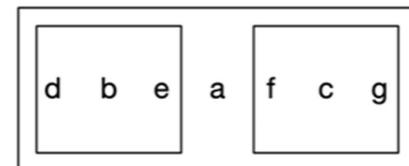
- **in-ordine:** Facciamo la chiamata ricorsiva sul nodo di sinistra, poi visitiamo la radice (passato alla funzione) e poi quello di destra.

```

1. Function inOrder(Node T)
2.   if T ≠ NIL then
3.     inOrder(T.left)
4.     visit(T)
5.     inOrder(T.right)

```

Costo (ottimo, pessimo, medio):  $\Theta(n)$



#### # breadth-first search (BFS, in ampiezza)

Vengono visitati tutti i nodi sullo stesso livello, partendo dall'insieme di nodi al livello 1, poi quelli al livello 2, etc. ... La BFS viene implementata tramite delle code.

Gli algoritmi ricorsivi sono quelli più adatti per la visita di alberi (a meno di vincoli imposti da superiori).

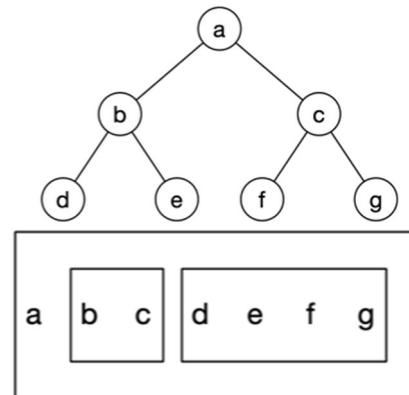
#### 5.1.2.1 Implementazione di BFS

```

1. function BFS(Tree T)
2.   Q = queue()
3.   if T.root ≠ NIL then
4.     enqueue(Q, T.root)
5.   while Q.size ≠ 0 do
6.     x = dequeue(Q)
7.     visit(x)
8.     if x.left ≠ NIL then
9.       enqueue(Q, x.left)
10.    if x.right ≠ NIL then
11.      enqueue(Q, x.right)

```

Costo (ottimo, pessimo, medio):  $\Theta(n)$



## 5.2 ALBERI BINARI DI RICERCA

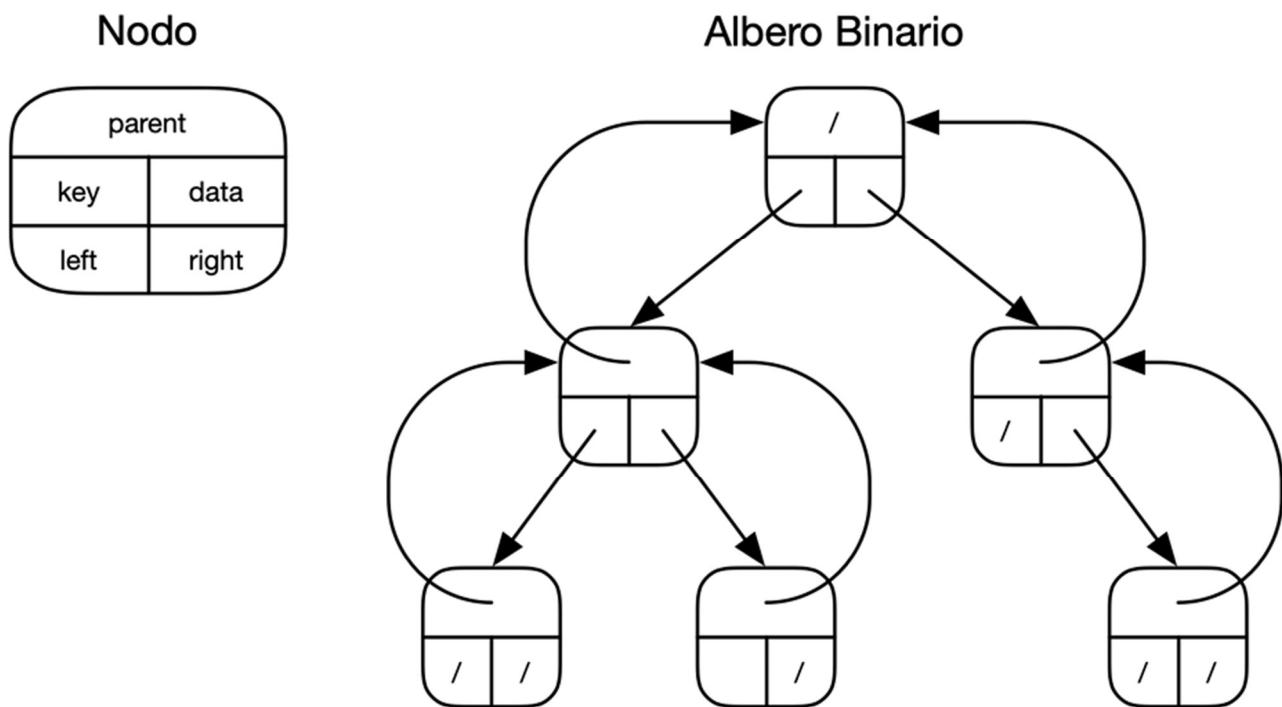
! IDEA: Portare la ricerca binaria sugli array negli alberi binari.

! IMPLEMENTAZIONE:

Ogni nodo  $v$  contiene due sottoalberi dove uno ha tutti i valori  $\leq$  (per esempio quello di destra) e l'altro ramo tutti i valori  $\geq$  (per esempio quello di sinistra).

La definizione va applicata ricorsivamente, dunque se avessimo un nodo con valore 10, e un sottoalbero con valore 6, che ha come sottонodi valori maggiori di 10 (per esempio 12) allora non sarebbe un albero valido per definizione.

Insomma la regola del maggiore/minore deve essere verificata in tutti gli alberi e *sottoalberi* dei nodi di destra/sinistra.

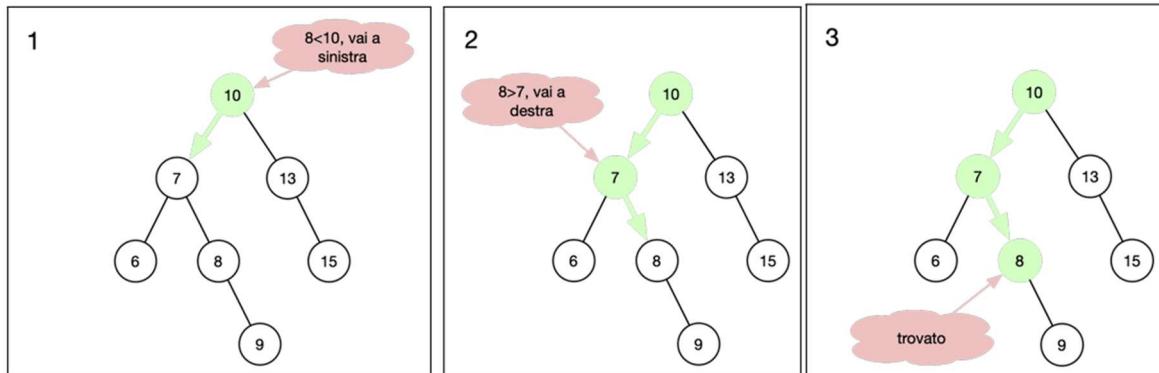


Operazioni fondamentali:

- `search(BST T, Key k)` Ritorna il nodo con chiave k in T oppure ritorna NIL
- `insert(BST T, Key k, Data d)` Inserisce in T (e ritorna) un nodo (k, d)
- `delete(BST T, Key k)` rimuove (e ritorna) il nodo k in T
  - o A supporto per delete abbiamo:
 

|                             |   |
|-----------------------------|---|
| <code>max(T)</code>         | ritorna il nodo con chiave massima k in T |
| <code>min(T)</code>         | ritorna il nodo con chiave minima k in T  |
| <code>predecessor(T)</code> | ritorna il nodo che precede T             |
| <code>successor(T)</code>   | <i>simmetrica a predecessor(T)</i>        |

### 5.2.1 Search



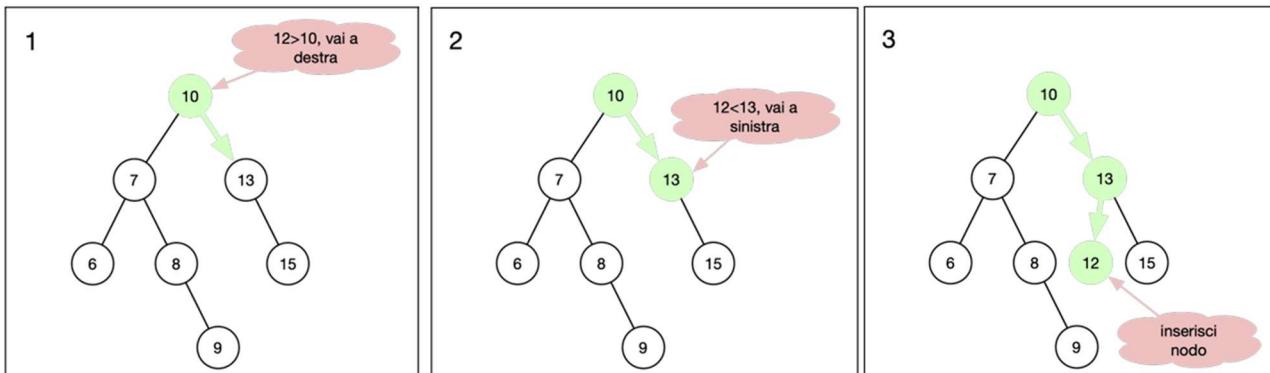
```

1. function search(BST T, Key k) → Node {
2.   tmp = T.root
3.   while tmp ≠ NIL do {
4.     if k == tmp.key then {
5.       return tmp
6.     } else if k < tmp.key then {
7.       tmp = tmp.left
8.     } else {
9.       tmp = tmp.right
10.    }
11.  }
12.  return NIL
13. }
```

Ritorna il primo nodo con chiave  $k$   
 Costo nel caso ottimo:  $O(1)$   
 Quando la radice ha chiave  $k$   
 Costo nel caso pessimo:  $\Theta(h)$   
 La visita è sempre confinata su di un percorso radice-foglia

$h$  = altezza albero

### 5.2.2 Insert



```

1. function insert(BST T, Key k, Data d) → Node {
2.   node = Node(k, d)
3.   prev = NIL
4.   curr = T.root
5.   while curr ≠ NIL do {
6.     prev = curr
7.     if k < curr.key then {
8.       curr = curr.left
9.     } else {
10.       curr = curr.right
11.     }
12.   }
13.   if prev == NIL then {
14.     T.root = node
15.   } else {
16.     node.parent = prev
17.     if k < prev.key then {
18.       prev.left = node
19.     } else {
20.       prev.right = node
21.     }
22.   }
23.   return node
24. }
```

Caso pessimo:  $\Theta(h)$   

- Ricerca posizione:  $\Theta(h)$
- Inserimento nodo:  $O(1)$

 Caso ottimo:  $O(1)$   

- Ricerca posizione:  $O(1)$
- Inserimento nodo:  $O(1)$

### 5.2.3 max e min

```

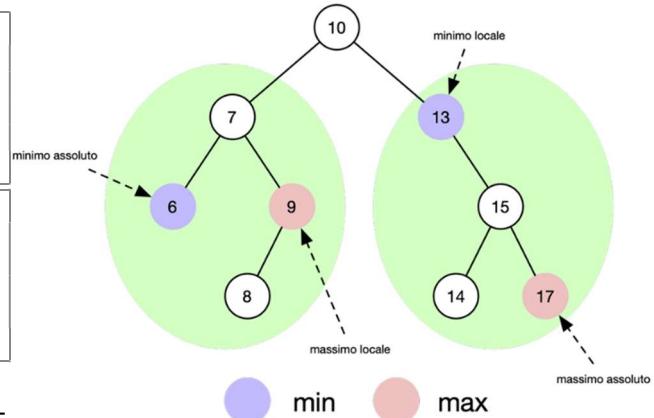
1. function max(Node T) → Node {
2.   while T ≠ NIL and T.right ≠ NIL do {
3.     T = T.right
4.   }
5.   return T
6. }
```

```

1. function max(Node T) → Node {
2.   while T ≠ NIL and T.left ≠ NIL do {
3.     T = T.left
4.   }
5.   return T
6. }
```

Dato un sottoalbero T

- Il nodo massimo in T è il nodo più a destra in T
- Il nodo minimo in T è il nodo più a sinistra



Stesso costo per entrambe le funzioni:

- Caso ottimo:  $O(1)$  (T non ha figli destri o sinistri)
- Caso pessimo:  $\Theta(h)$

N.B.: Non confrontiamo chiavi, usiamo solo la struttura dell'albero

### 5.2.4 Predecessor

Il predecessore di un nodo  $v$  è il nodo  $u$  che precede  $v$  quando i nodi sono ordinati rispetto a una visita in-ordine

Caso 1:

- Il nodo  $v$  ha un figlio sinistro
- Il predecessore è il nodo  $u$  con chiave massima nel sottoalbero sinistro di  $v$
- Per la proprietà di ordine dei BST il sottoalbero sinistro di  $v$  contiene solo chiavi  $\leq v.key$

Caso 2:

- Il nodo  $v$  **non** ha un figlio sinistro
- Il predecessore è il nodo  $u$  con chiave massima nel sottoalbero sinistro di  $v$
- Per la proprietà di ordine dei BST il nodo  $v$  è il nodo minimo nel sottoalbero destro di  $u$

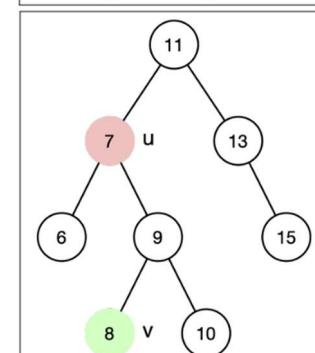
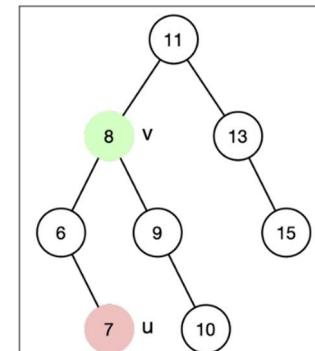
successor è simmetrica a predecessor

```

1. function predecessor(Node T) → Node {
2.   if T == NIL then {
3.     return NIL
4.   } else if T.left ≠ NIL then {
5.     Return max(T.left)
6.   } else {
7.     P = T.parent
8.     while P ≠ NIL and T == P.left do {
9.       T = P
10.      P = P.parent
11.    }
12.    return P
13. }
```

Caso pessimo:  $\Theta(h)$

- Caso 1 (max):  $\Theta(h)$
- Caso 2:  $\Theta(h)$



Caso Ottimo:  $O(1)$

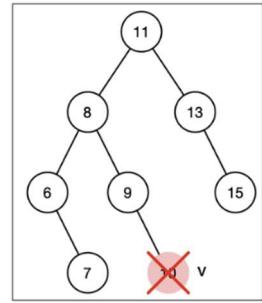
- Caso 1 (max):  $O(1)$
- Caso 2:  $O(1)$

N.B.: Non confrontiamo chiavi, usiamo solo la struttura dell'albero

### 5.2.5 delete

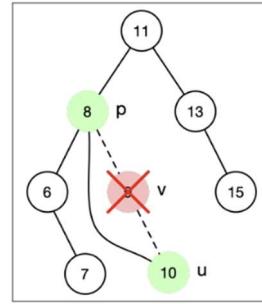
Caso 1:

- Il nodo  $v$  da rimuovere è una foglia
- Semplicemente rimuoviamo  $v$
- Se rimuoviamo una foglia non alteriamo la proprietà di ordine dei BST nei nodi rimanenti



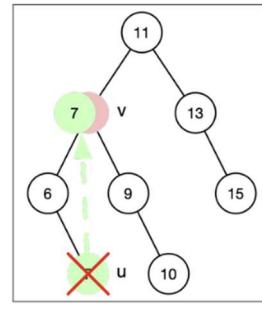
Caso 2:

- Il nodo da rimuovere  $v$  ha un solo figlio  $u$
- $u$  diventa figlio del genitore  $p$  di  $v$
- Se  $v$  è figlio sinistro,  $u$  diventa figlio sinistro di  $p$  (e viceversa per dx)
- Possiamo rimuovere  $v$
- Per la proprietà di ordine dei BST, se  $v$  è un figlio destro tutte le chiavi nel sottoalbero radicato in  $u$  sono  $\geq p.key$  e se  $v$  è un figlio sinistro tutte le chiavi in  $u$  sono  $\leq p.key$



Caso 3:

- Il nodo da rimuovere  $v$  ha due figli
- Cerchiamo il predecessore  $u$  di  $v$
- Copiamo  $v.key = u.key$  (e dati)
- Il nodo  $u$  ha al massimo un figlio
- Rimuoviamo il nodo  $u$  (Caso 1 o 2)
- Per la proprietà di ordine dei BST, la chiave  $u.key$  del predecessore di  $v$  è  $\geq$  di tutte le chiavi in  $v.left$  e  $\leq$  di tutte le chiavi in  $v.right$ . Possiamo quindi sostituire  $v$  con  $u$  senza alterare la proprietà di ordine dei BST



```

1. function delete(BST T, Key k) → Node {
2.   v = search(T, k)
3.   if v ≠ NIL then {
4.     if v.left ≠ NIL and v.right ≠ NIL then {
5.       u = predecessor(v)
6.       swap(v, u)
7.       v = u
8.     }
9.     disconnect(T, v)
10.  }
11.  return v
12. }
```

La procedura `delete` rimuove il nodo dall'albero ma non azzera i riferimenti ai figli e al padre

```

1. function disconnect(BST T, Node v) {
2.   p = v.parent
3.   if v.right == NIL then {
4.     c = v.left
5.   } else {
6.     c = v.right
7.   }
8.   if p == NIL then {
9.     T.root = c           // v era nodo root
10.  } else if p.left == v then {
11.    p.left = c          // v era figlio sx
12.  } else {
13.    p.right = c         // v era figlio dx
14.  }
15.  if c ≠ NIL then {
16.    c.parent = p
17.  }
18. }
```

La procedura `disconnect` gestisce solo i casi 1 e 2 ed eventualmente aggiorna la radice dell'albero

- I casi 1 e 2 richiedono un numero costante di operazioni
- $p$  punta al padre di  $v$  mentre  $c$  punta al suo eventuale (unico) figlio
- Notare che il campo padre del nodo  $v$  non viene modificato

Caso pessimo:  $\Theta(h)$

- search:  $\Theta(h)$
- disconnect:  $O(1)$
- predecessor:  $\Theta(h)$

Caso ottimo:  $O(1)$

- search:  $O(1)$
- disconnect:  $O(1)$
- predecessor:  $O(1)$

Caso Medio;

Nel caso pessimo tutte le operazioni su BST hanno costo  $\Theta(h)$

- il costo medio dipende dall'altezza media  $\bar{h}$  di un BST

L'altezza  $h$  di un BST può variare di molto

- L'altezza massima di un BST con  $n$  nodi è  $h = \Theta(n)$

- Un BST con altezza  $h$  ha almeno  $h + 1$  nodi
- Quando l'albero binario è una lista

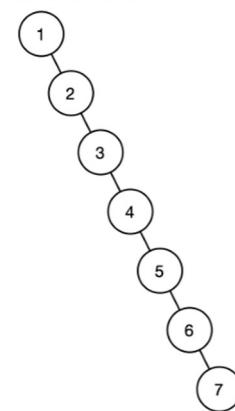
- l'altezza minima di un BST con  $n$  nodi è  $h = \Theta(\log n)$

- Un BST con altezza  $h$  ha al massimo  $2^{h+1} - 1$  nodi
- Quando l'albero binario è *perfetto*

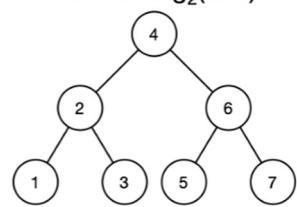
Qual è l'altezza media  $\bar{h}$  di un BST?

- È possibile dimostrare che  $\bar{h} = O(\log n)$

$n=7 h=n-1=6$



$n=7 h=\log_2(n+1)-1 = 2$



### 5.3 ALBERI AVL (ADELSON-VELSKY, LANDIS)

Un albero AVL è un albero binario (quasi) perfettamente bilanciato.

L'obiettivo è avere operazioni d'inserimento e rimozione autobilanciati con costo pessimo di  $O(\log n)$ .

› **Fattore di bilanciamento:**

Indicato con  $\beta(v)$  di un nodo  $v$ , è dato dalla differenza tra l'altezza del sottoalbero sinistro e del sottoalbero destro di  $v$ :

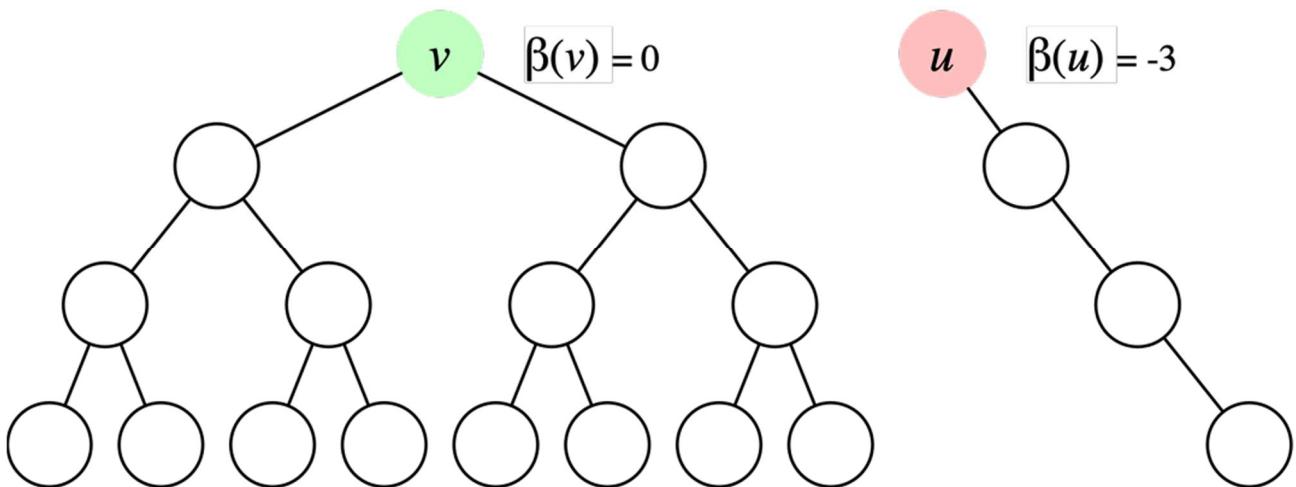
$$\beta(v) = \text{altezza}(v.\text{left}) - \text{altezza}(v.\text{right})$$

› **Bilanciamento in altezza:**

Un albero si dice bilanciato in altezza se le altezze dei sottoalberi sinistro e destro in ogni nodo differiscono al massimo di 1.

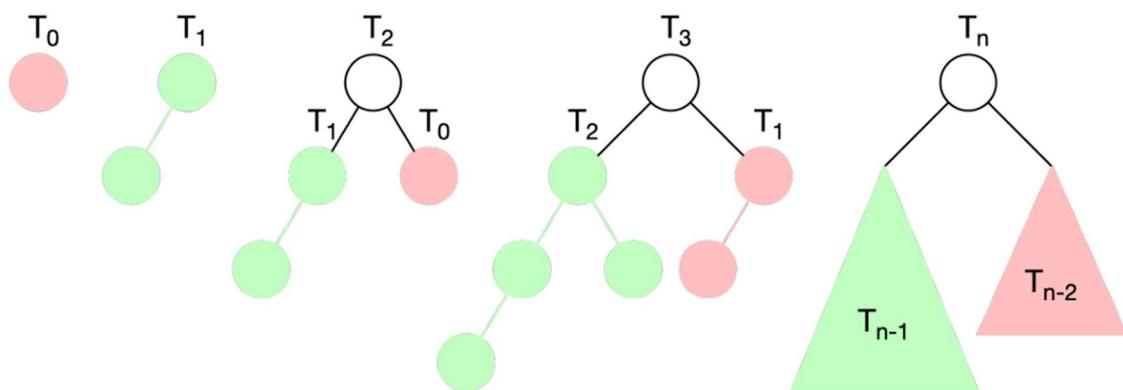
In altre parole, un albero è bilanciato in altezza se  $\forall v \in \text{Nodi}, |\beta(v)| \leq 1$

Fattore di bilanciamento:



#### Qual è l'altezza massima di un albero AVL?

Proviamo a costruire *alberi di Fibonacci*, ossia alberi che hanno in ogni nodo (foglie escluse)  $|\beta(v)| = 1$ , ossia ogni albero (di sinistra) ha profondità +1 rispetto a quello (di destra).



Questa è la configurazione più sbilanciata che possiamo ottenere continuando a aderire alle regole degli AVL.

Per costruzione si ha che il numero di nodi ( $n$ ) dell' $h$ -esimo sottoalbero ha valore

$$n_h = n_{h-1} + n_{h-2} + 1$$

Quindi, un albero di Fibonacci di altezza  $h$  ha

$$n_h = F_{h+3} - 1 \text{ nodi}$$

Ricordiamo che

$$F_h = \Theta(\phi^h) \text{ dove } \phi \approx 1.618$$

Da cui otteniamo che

$$n_h = F_{h+3} - 1 = \Theta(\phi^{h+3}) = \Theta(\phi^h)$$

Equivalentemente

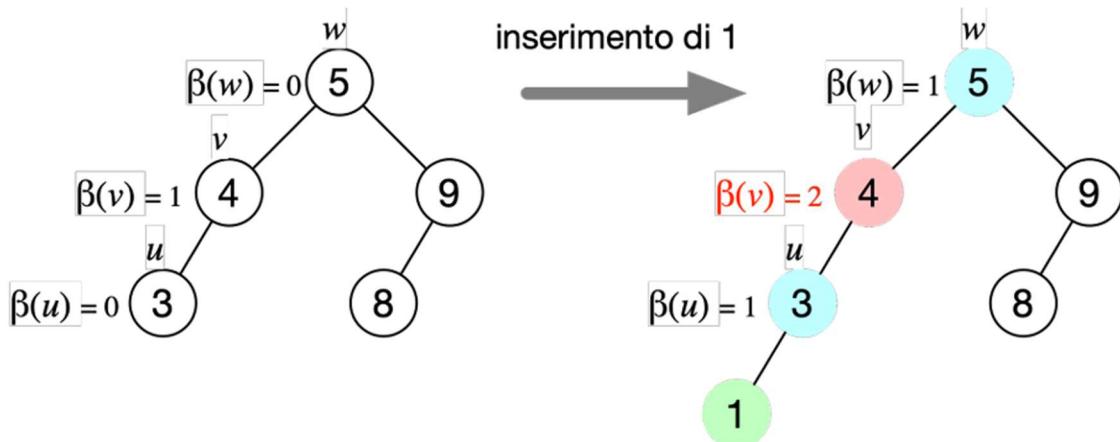
$$\log_\phi n_h = \Theta(\log_\phi \phi^h) \Rightarrow \log_\phi n_h = \Theta(h)$$

E possiamo quindi concludere che

$$h = \Theta(\log n_h)$$

### Mantenere il bilanciamento

La ricerca su un albero AVL viene effettuata come su di un BST, inserimenti e rimozioni richiedono di essere modificati per mantenere il bilanciamento dell'albero.



Per poter verificare il bilanciamento ogni nodo  $u$  deve mantenere informazioni sull'altezza del proprio sottoalbero  $u.height$ , serve per poter calcolare il fattore di bilanciamento  $\beta$ .

### Aggiornamento altezza e bilanciamento

```

1. function updateHeight(Node u)
2.   if u ≠ NIL then
3.     lefth = -1; righth = -1;
4.     if u.left ≠ NIL then
5.       lefth = u.left.height;
6.     if u.right ≠ NIL then
7.       righth = u.right.height;
8.     u.height = Max(lefth, righth) + 1

```

```

1. function balanceFactor(Node u) → Int
2.   lefth = -1;
3.   righth = -1;
4.   if u ≠ NIL and u.left ≠ NIL then
5.     lefth = p.left.height;
6.   if u ≠ NIL and u.right ≠ NIL then
7.     righth = p.right.height;
8.   return lefth-righth;

```

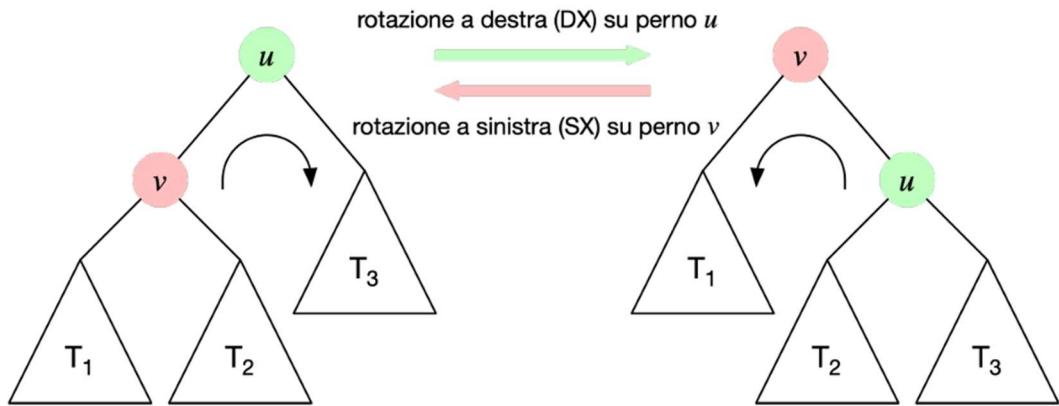
Entrambe costano  $O(1)$

`updateHeight` aggiorna l'altezza del nodo  $u$

`balanceFactor` calcola il fattore di bilanciamento  $\beta$  del nodo  $u$

### Rotazioni

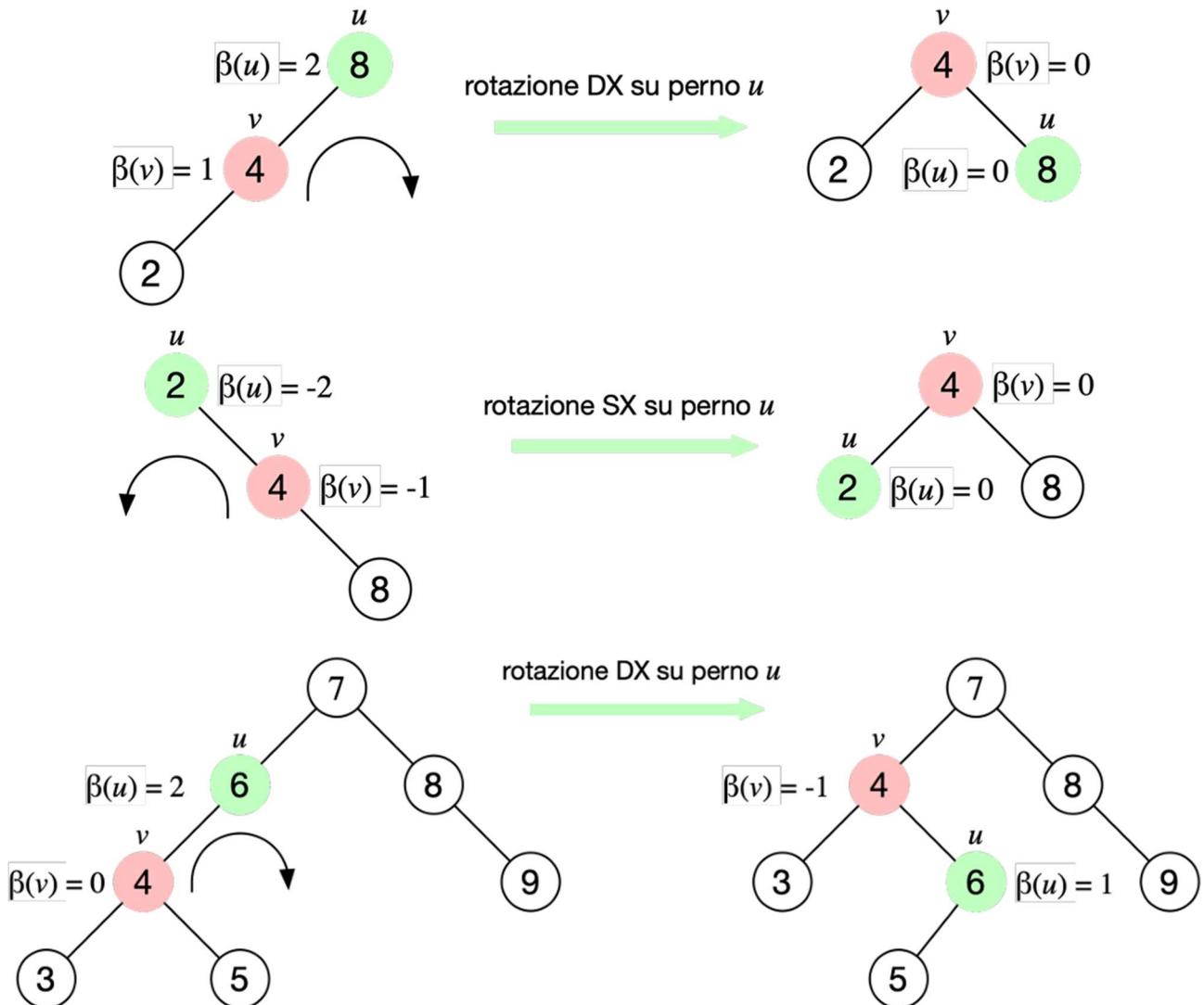
Rotazione semplice: operazione fondamentale per ribilanciare l'albero



Preserva la proprietà di ordine dei BST

- La chiave  $u.key \geq v.key$  e maggiore o uguale di ogni chiave in  $T_1, T_2$
- La chiave  $v.key \leq u.key$  e minore o uguale di ogni altra chiave in  $T_2, T_3$

Una rotazione semplice modifica solo il fattore di bilanciamento di  $u, v$



### Pseudocodice: Rotazione a Destra

```

1. function rightRotate(AVL T, Node u)
2.   if u ≠ NIL and u.left ≠ NIL then
3.     v = u.left
4.     u.left = v.right
5.     v.right = u
6.     if u.left ≠ NIL then
7.       u.left.parent = u
8.     v.parent = u.parent
9.     if u.parent == NIL then      // u era il nodo radice
10.      T.root = v             // v è la nuova radice
11.    else if u.parent.left == u then // u era un figlio sinistro
12.      u.parent.left = v
13.    else                      // u era un figlio destro
14.      u.parent.right = v
15.    updateHeight(u)
16.    updateHeight(v)
17.

```

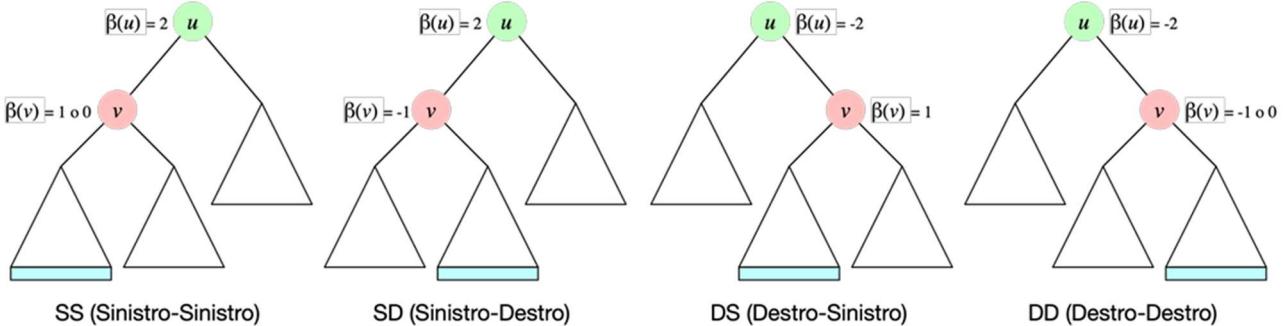
- Costo:  $O(1)$
- leftRotate è simmetrica

### Sbilanciamenti

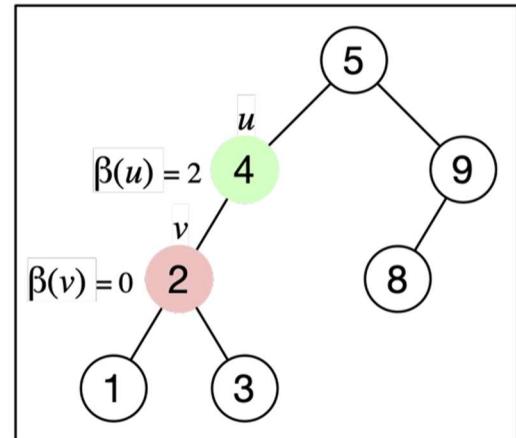
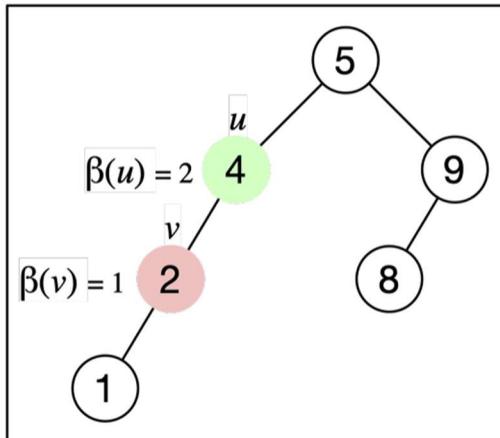
Assumiamo di avere un albero AVL bilanciato, in cui il sottoalbero radicato in un nodo  $u$  diventa sbilanciato in seguito a una operazione di inserimento o rimozione.

Abbiamo quattro casi (simmetrici a due a due)

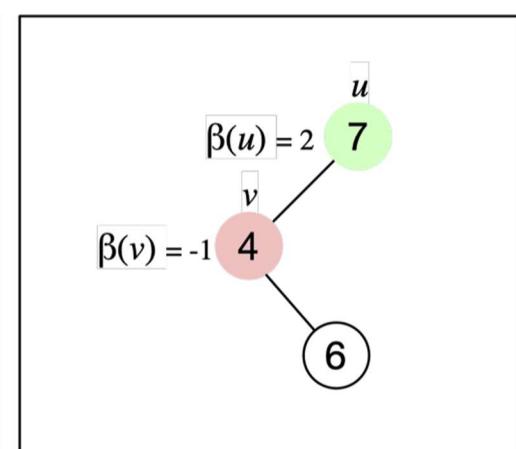
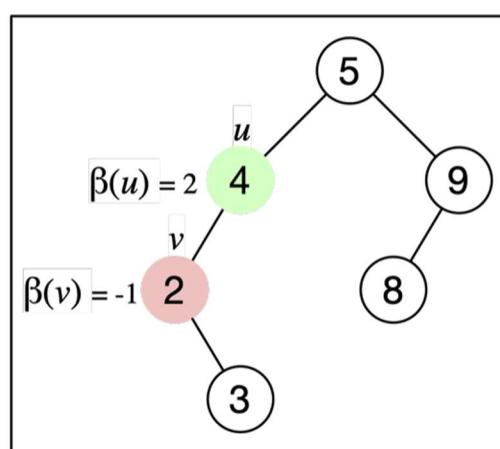
1. Sbilanciamento SS:  $\beta(u) = 2$  e  $\beta(u.left) \geq 0$
2. Sbilanciamento DD:  $\beta(u) = -2$  e  $\beta(u.right) \leq 0$
3. Sbilanciamento SD:  $\beta(u) = 2$  e  $\beta(u.left) = -1$
4. Sbilanciamento DS:  $\beta(u) = -2$  e  $\beta(u.right) = 1$



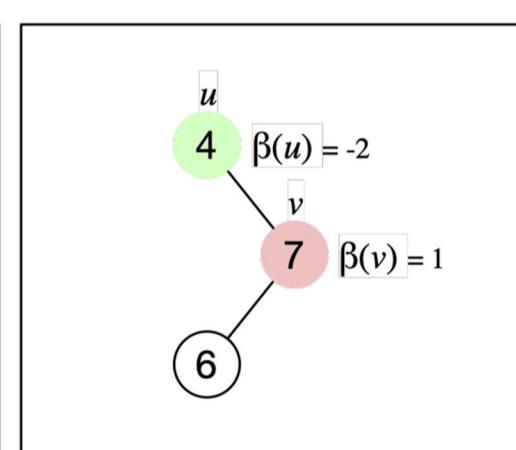
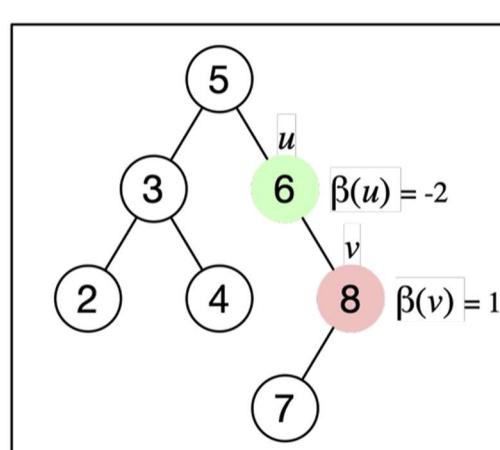
**ESEMPI:**  
Sbilanciamenti SS



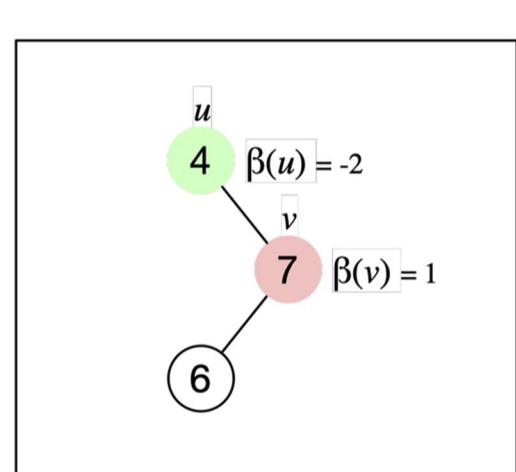
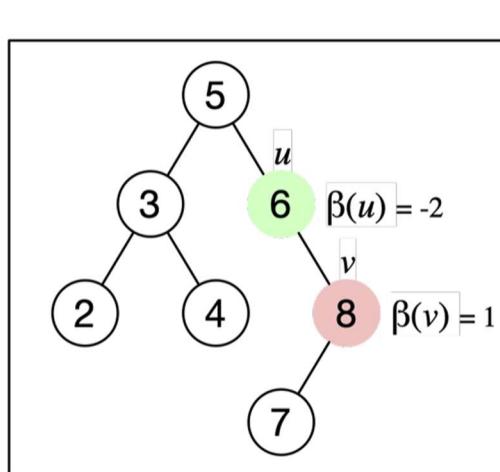
## Sbilanciamenti SD



## Sbilanciamenti DD



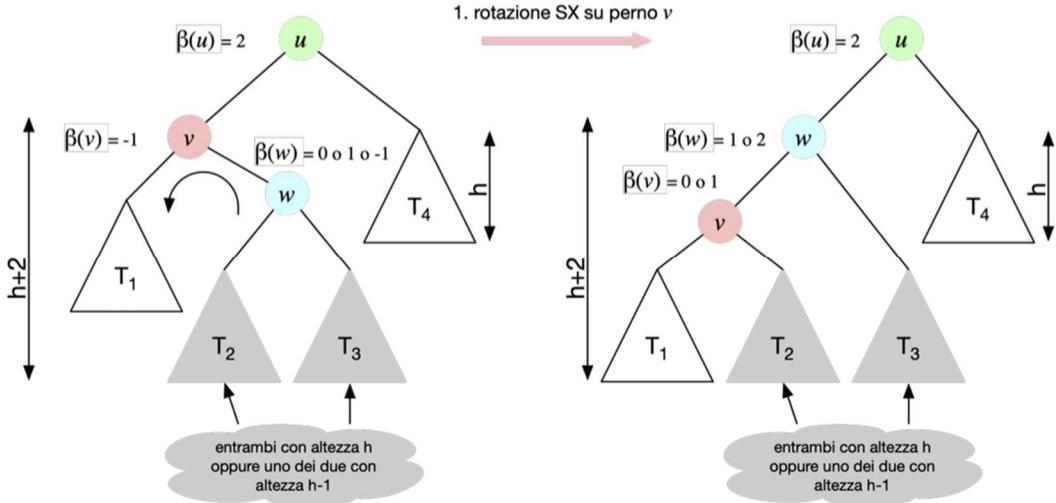
## Sbilanciamenti DS



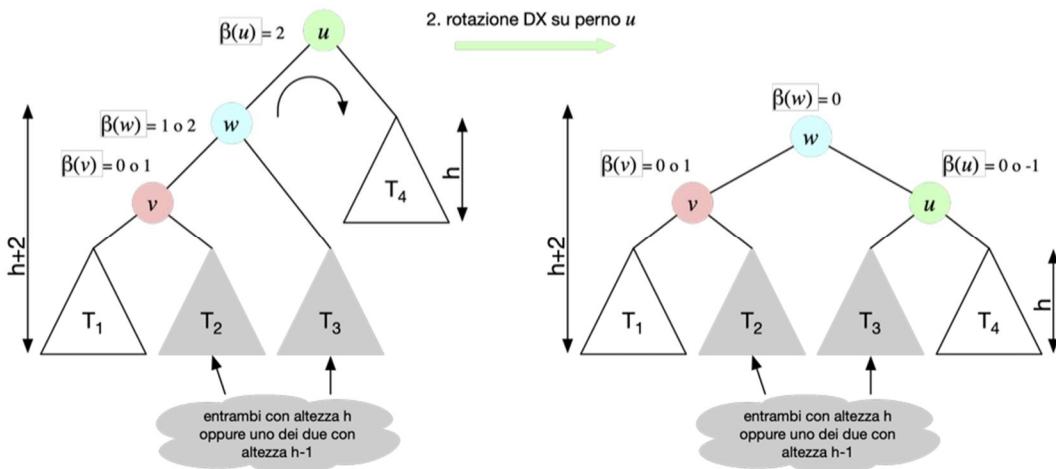
Ribilanciamento per uno sbilanciamento SD  $\Rightarrow$  Rotazione DX (non funziona, vedi slide)  
 $\Rightarrow$  Rotazione SXDX

$$\beta(u) = 2 \quad \text{e} \quad \beta(u.left) = -1$$

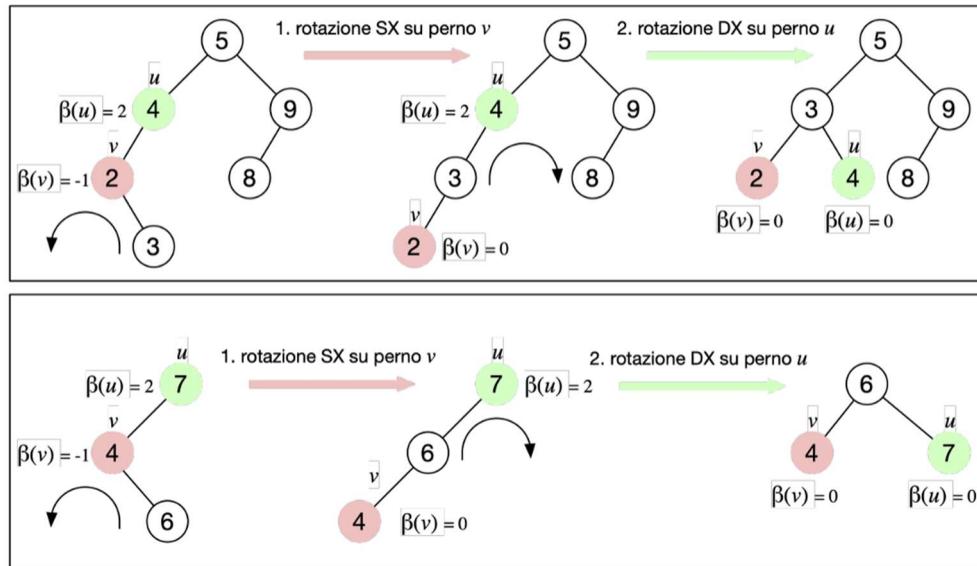
**Step 1:** Rotazione SX con perno  $u.left \Rightarrow$  Sbilanciamento SS



**Step 2:** Rotazione DX con perno  $u$



**Esempio:**



## Riassunto: Sbilanciamenti e Rotazioni

| Tipo di Sbilanciamento        | Condizione di Sbilanciamento              | Rotazioni Necessarie                               |
|-------------------------------|---|--|
| <b>SS (Sinistro-Sinistro)</b> | $\beta(u) = -2$ e $\beta(u.left) \geq 0$  | Rotazione DX su $u$                                |
| <b>DD (Destro-Destro)</b>     | $\beta(u) = -2$ e $\beta(u.right) \leq 0$ | Rotazione SX su $u$                                |
| <b>SD (Sinistro-Destro)</b>   | $\beta(u) = 2$ e $\beta(u.left) = -1$     | Rotazione SX su $u.left$ , poi Rotazione DX su $u$ |
| <b>DS (Destro-Sinistro)</b>   | $\beta(u) = 2$ e $\beta(u.right) = 1$     | Rotazione DX su $u.right$ poi Rotazione SX su $u$  |

### Legenda:

- $\beta(u)$ : Fattore di bilanciamento del nodo  $u$ .
- $u.left$ : Figlio sinistro del nodo  $u$ .
- $u.right$ : Figlio destro del nodo  $u$ .
- Rotazione DX: Rotazione a destra.
- Rotazione SX: Rotazione a sinistra.

### Pseudocodice: Bilanciamento

```

1. function balance(AVL T, Node u) {
2.   b = balanceFactor(u)
3.   if b == 2 then {                                // SS o SD
4.     if balanceFactor(u.left) == -1 then {
5.       leftRotate(T, u.left)                      // SD
6.     }
7.     rightRotate(T, u)
8.   }
9.   else if b == -2 then {                           // DD o DS
10.    if balanceFactor(u.right) == 1 then {
11.      rightRotate(T, u.right)                     // DS
12.    }
13.    leftRotate(T, u)
14.  }
15. }
```

Costo:  $O(1)$

### Pseudocodice: Insert

```

1. function insert(AVL T, Key k, Data d) → Node
2.   v = BSTInsert(T, k, d)
3.   p = v.parent
4.   while p ≠ NIL and |balanceFactor(p)| ≠ 2 do
5.     updateHeight(p)
6.     p = p.parent
7.     if p ≠ NIL then
8.       balance(T, p)
9.   return v
```

- **BSTInsert** costa in ogni caso  $\Theta(\log n)$ 
  - > l'altezza dell'albero AVL è  $\Theta(\log n)$
  - > Inoltre, l'albero è sempre bilanciato
- il ciclo while (r4) viene eseguito  $O(\log n)$  volte
  - > al peggio, per tutti i nodi sul percorso  $v$  fino alla radice

### Dettagli:

1. Si inserisce il nuovo nodo come per i BST
  - Tutti i percorsi sono lunghi  $\Theta(\log n)$

⇒ Costo:  $\Theta(\log n)$
2. Si riaggiorano le altezze dei sottoalberi, eventualmente cambiate
  - Nel caso peggiore l'aggiornamento va fatto per tutti i nodi fino alla radice ⇒ Costo:  $O(\log n)$
3. Se un nodo presenta un fattore di bilanciamento  $\pm 2$  (nodo critico), occorre ribilanciare l'albero con la procedura di ribilanciamento
  - In caso di inserimento abbiamo al più un nodo critico
  - Dopo un ribilanciamento non continuiamo ad aggiornare altezze ⇒ Costo pessimo:  $O(1)$

Costo inserimento in ogni caso:  $\Rightarrow \Theta(\log n)$

### Pseudocodice: Delete

```

1. function delete(AVL T, Key k, Data d) {
2.   v = BSTDelete(T, k)
3.   if v ≠ NIL then {
4.     p = v.parent
5.     while p ≠ NIL do {
6.       if |balanceFactor(p)| == 2 then {
7.         balance(T, p)
8.       } else {
9.         updateHeight(p)
10.      }
11.      p = p.parent
12.    }
13.  }
14.  return v
15. }
```

- `BSTDelete` è la procedura `delete` su Alberi Binari

> Ricordiamo che `delete` non azzerà il campo `parent` del nodo rimosso `v`, quindi l'assegnamento a riga 4 è corretto

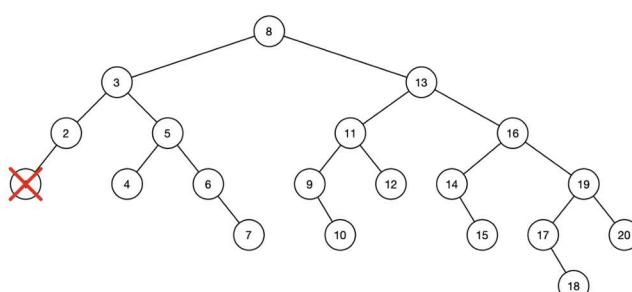
- Notare che `balance` può essere richiamata molte volte

Dettagli:

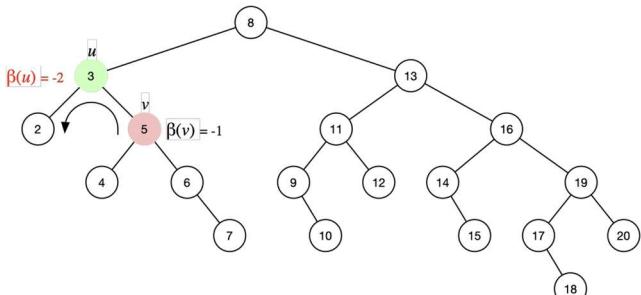
1. Si rimuove il nodo come per i BST
  - Costo in caso pessimo e medio  $\Rightarrow \Theta(\log n)$
2. Si riaggiorano le altezze dei sottoalberi, eventualmente cambiate
  - Nel caso peggiore l'aggiornamento va fatto per tutti i nodi fino alla radice
  - Costi:      caso pessimo  $\Theta(\log n)$
  - Costi:      caso medio  $O(\log n)$
3. Se un nodo presenta un fattore di bilanciamento  $\pm 2$  (nodo critico), occorre ribilanciare l'albero con la procedura di ribilanciamento
  - In caso di rimozione potremmo avere più nodi critici
  - Tutti i nodi critici sono in un unico percorso radice-nodo rimosso
  - Costi:      caso pessimo  $\Theta(\log n)$
  - Costi:      caso medio  $O(\log n)$

Costo medio/pessimo della rimozione:  $\Rightarrow \Theta(\log n)$

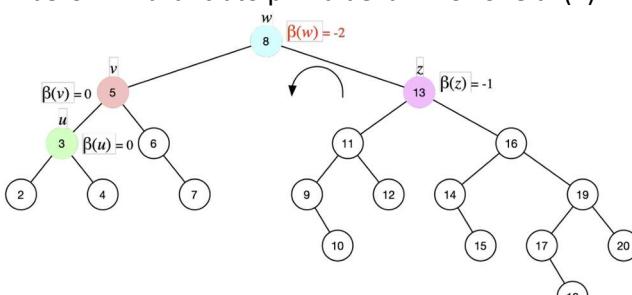
### ESEMPIO: ROTAZIONI A CASCATA



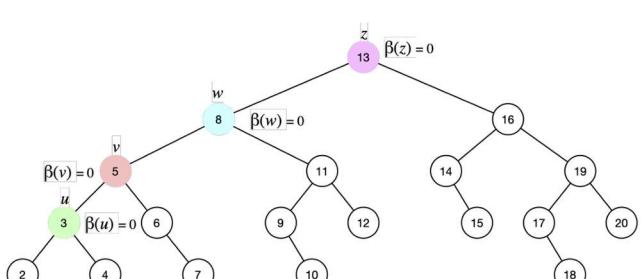
Albero AVL bilanciato prima della rimozione di (1)



Ribilanciamento: rotazione SX con perno  $u$



Ribilanciamento: rotazione SX con perno  $w$



Albero ribilanciato

Gli alberi AVL non sono l'unica struttura dati auto-bilanciante, altre implementazioni non richiedono necessariamente alberi binari e tra le strutture dati più note abbiamo:

- B-alberi
  - Alberi generici non necessariamente binari, utilizzati per database e file system
- Alberi 2-3
  - Ogni nodo intero ha 2 o 3 nodi
  - Tutte le foglie sono sempre allo stesso livello
  - Tutte le chiavi sono ordinate nelle foglie
- RB-Alberi o Red-Black Trees
  - Alberi binary come gli AVL

Supportano tutte l'inserimento, rimozione e ricerca in tempo logaritmico

#### Riassunto:

|                           | search            |                  | insert            |                  | delete            |                  |
|---------------------------|-------------------|------------------|-------------------|------------------|-------------------|------------------|
|                           | Medio             | Pessimo          | Medio             | Pessimo          | Medio             | Pessimo          |
| Array non ordinati        | $\Theta(n)$       | $\Theta(n)$      | $\Theta(n)$       | $\Theta(n)$      | $\Theta(n)$       | $\Theta(n)$      |
| Array ordinati            | $O(\log n)$       | $\Theta(\log n)$ | $\Theta(n)$       | $\Theta(n)$      | $O(n)$            | $\Theta(n)$      |
| Lista concatenata         | $\Theta(n)$       | $\Theta(n)$      | $\Theta(n)$       | $\Theta(n)$      | $\Theta(n)$       | $\Theta(n)$      |
| Albero Binario di ricerca | $\Theta(\bar{h})$ | $\Theta(h)$      | $\Theta(\bar{h})$ | $\Theta(h)$      | $\Theta(\bar{h})$ | $\Theta(n)$      |
| Albero AVL                | $\Theta(\log n)$  | $\Theta(\log n)$ | $\Theta(\log n)$  | $\Theta(\log n)$ | $\Theta(\log n)$  | $\Theta(\log n)$ |

- $h$  = altezza albero,  $\bar{h}$  = altezza media albero
- L'implementazione di un dizionario con alberi AVL richiede unicamente la modifica dell'operazione di inserimento per evitare chiavi ripetute
- Gli alberi AVL assicurano un costo logaritmico per tutte le operazioni.

## 6 TABELLE HASH

Molte applicazioni richiedono una struttura dati che supporti in maniera efficiente *solo* le operazioni basilari: `search`, `insert`, `delete`

La **Tabella Hash** fa al caso nostro per queste situazioni,

Indichiamo con:

- #  $U$  = Universo delle chiavi possibili
- #  $K$  = Insieme delle chiavi effettivamente memorizzate

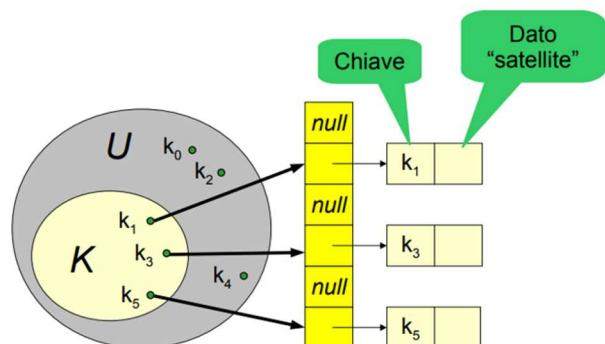
A seconda di come sono implementati questi due assieme si avranno tipi di strutture dati differenti, ad esempio:

- $(K \sim U) \rightarrow U$  corrisponde oppure è simile a  $K$   
Usiamo **tabelle ad indirizzamento diretto**
- $(K \ll U) \rightarrow U$  è un insieme generico molto più grande di  $K$   
Usiamo **Tabelle Hash**

### 6.1.1 Tabelle ad indirizzamento diretto

L'implementazione è basata su array, dove l'elemento con chiave  $k$  è memorizzato nel  $k$ -esimo "slot" dell'array. Il dato corrispondente alla cella verrà chiamato "dato satellite"

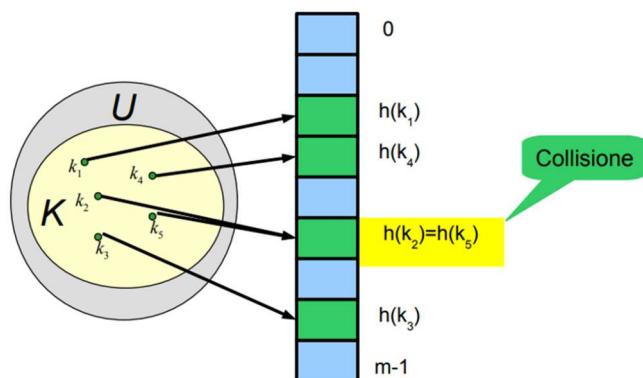
È facile quindi intuire che se  $|K| \sim |U|$  non sprechiamo troppo spazio e le operazioni saranno eseguite in tempo  $O(1)$  nel caso peggiore.



Invece se abbiamo  $|K| \ll |U|$ , questo tipo di implementazione non è praticabile. Per questo ci riconduciamo alle tabelle Hash.

## 6.2 TABELLE HASH

Questo tipo di struttura dati ha due elementi fondamentali che la compone:



- Una tabella (array)  $A[m]$   
 $A := (A[0], \dots, A[m - 1])$
- Una funzione Hash  
 $h := U \rightarrow \{0, \dots, m - 1\}$

Prendiamo una chiave  $k$ , definiamo il valore hash della chiave come  $h(k)$ . Quindi la funzione hash  $h$  trasforma una chiave in un indice della tabella  $A$ . La chiave  $k$  viene "mappata" nello slot  $A[h(k)]$ .

Quando due chiavi diverse hanno lo stesso valore

hash, avviene una **collisione**. Idealmente vogliamo funzioni hash senza collisioni.

Per realizzare una tabella hash efficiente abbiamo bisogno di:

- Un vettore
- Una funzione hash calcolabile velocemente e che garantisca una "buona" distribuzione delle chiavi nel vettore
- Un meccanismo per gestire le collisioni

### 6.2.1 Problema delle collisioni

Il problema della collisione non è presente se si usa una funzione hash perfetta

Una funzione hash si dice **perfetta** se è *iniettiva*, cioè  $\forall u, v \in U : u \neq v \rightarrow h(u) \neq h(v)$

Il problema di avere una funzione hash perfetta è che lo spazio delle chiavi è spesso grande e sparso; quindi, possiamo dire che una funzione hash perfetta è impossibile da ottenere.

Di conseguenza ci saranno sicuramente delle collisioni.

## 6.3 FUNZIONI HASH

Visto che le collisioni sono inevitabili, cerchiamo di diminuire il più possibile: vogliamo funzioni che distribuiscano *uniformemente* le chiavi negli indici  $[0, \dots, m - 1]$  della tabella hash.

### 6.3.1 Uniformità semplice

Una buona funzione hash che soddisfa la proprietà di

- La funzione deve distribuire uniformemente le chiavi negli indici dell'array  $A$
- Ogni indice  $i = h(k)$  deve essere generato con probabilità  $\frac{1}{m}$
- Se alcuni  $i \in [0, \dots, m - 1]$  vengono scelti con maggiore probabilità da  $h$ , allora avremo un numero maggiore di collisioni

### 6.3.2 Assunzioni

Tutte le chiavi sono equiprobabili:  $P(k) = \frac{1}{|U|}$  (necessario semplificare per porre un meccanismo generale)

Le chiavi sono valori numerici non negativi:

È sempre possibile trasformare una chiave complessa in un numero, ad esempio, considerando il valore decimale della sua rappresentazione binaria.

La funzione hash può essere calcolata in tempo  $O(1)$ .

### 6.3.3 Funzione da stringa a valori numerici

Concateniamo la rappresentazione binaria dei caratteri che compongono la stringa.

Dobbiamo quindi mappare l'alfabeto inglese:  $a = 1, b = 2, c = 3, \dots, z = 26$ , quindi sono sufficienti 6 bit per carattere

$$\text{bin}(\text{"beer"}) = \underline{000010} \quad \underline{000101} \quad \underline{000101} \quad \underline{010010}$$

545.106 in  
base 10

### 6.3.4 Divisione

Il metodo della divisione è basato sul resto della divisione per  $m$ :  $h(k) = k \% m$

Esempio:  $m = 12, k = 100 \Rightarrow h(k) = 4$  Il vantaggio principale è la velocità di esecuzione.

Però è suscettibile a specifici valori di  $m$ , se  $m = 2^p \Rightarrow h(k)$  dipende unicamente dai  $p$  bit meno significativi di  $k$  e non da tutti i bit di  $k$ .

Soluzione: scegliere  $m$  come numero primo non troppo vicino ad una potenza di 2.

### 6.3.5 Moltiplicazione

Il metodo della moltiplicazione si basa sulla seguente formula:

$$h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

(le parentesi strane prendono la parte intera del numero)

Sia  $A$  una costante compresa tra 0 e 1. Moltiplichiamo  $k$  per  $A$  e prendiamo la parte frazionaria. Moltiplichiamo quest'ultima per  $m$  e prendiamo la parte intera.

È più lento del metodo della divisione, ma ha il vantaggio che  $m$  non è critico.

Il valore di  $A$  viene scelto usando Knuth  $\rightarrow A \sim (\sqrt{5} - 1)/2 = 0.61803 \dots$

### 6.3.6 Codifica algebrica

Il metodo della codifica algebrica si basa sulla seguente formula:

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + \dots + k_1 x + k_0) \bmod m$$

- #  $k = k_n \cdot k_{n-1} \cdots k_1 \cdot k_0$  è l' $i$ -esimo bit della rappresentazione binaria di  $k$ , oppure l' $i$ -esima cifra della rappresentazione decimale di  $k$ , o anche il codice ascii dell' $i$ -esimo carattere

Ha il vantaggio che dipende da tutti i bit/caratteri della chiave. Ha lo svantaggio di avere  $n$  addizioni e  $\frac{n(n-1)}{2}$  prodotti da eseguire.

### 6.3.7 Regola di Horner

Consiste nella valutazione di un polinomio in un punto. Un polinomio di grado  $n$  può essere riscritto valutato calcolando  $n$  addizioni e  $n$  prodotti (invece di  $\frac{n(n-1)}{n}$  prodotti)

$$p(x) = a_n x^n + \dots + a_0 = a_0 + x(a_1 + x(a_2 + x(\dots + x(x_{n-1} + a_n x) \dots)))$$

## 6.4 RISOLUZIONE DELLE COLLISIONI

Abbiamo ridotto, ma non eliminato, il numero di collisioni. Ora dobbiamo trovare un modo per gestire le collisioni residue.

Dobbiamo trovare collocazioni alternative per le chiavi; se una chiave non si trova nella posizione attesa, bisogna andare a cercare nelle posizioni alternative (le operazioni possono costar  $\Theta(n)$  nel caso pessimo).

Abbiamo due tecniche possibili:

### 6.4.1 Concatenamento (o scansione esterna)

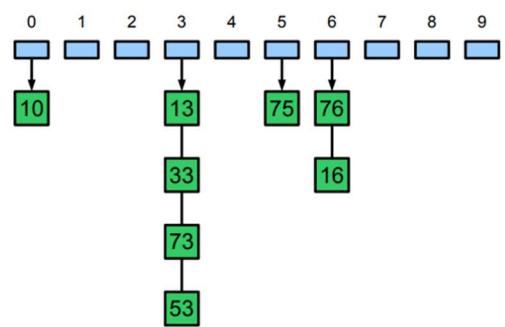
Memorizza gli elementi con lo stesso valore di hash in una lista concatenata. Le operazioni e la gestione è quindi quella di una lista.

Le operazioni hanno un costo computazionale che dipende dal **fattore di carico**:  $\alpha = \frac{n}{m}$ , dove  $n$  è il numero degli elementi memorizzati e  $m$  la dimensione della tabella.

**CASO PESSIMO:** (quando tutte le chiavi sono in una lista unica)

- Insert:  $\Theta(1)$
- Search:  $\Theta(n)$
- Delete:  $\Theta(n)$

$$h(k) = k \bmod 10$$



## CASO MEDIO

La complessità delle operazioni può essere rappresentata dal numero medio di accessi per cercare (con successo o insuccesso) una chiave.

Quindi il caso medio dipende da come le chiavi vengono distribuite; assumiamo un hashing uniforme semplice, da cui ogni slot della tabella avrà mediamente  $\alpha$  chiavi.

Ricerca con insuccesso:

*Teorema:* in tabella hash con concatenamento, una ricerca senza successo richiede un tempo atteso di  $\Theta(1 + \alpha)$

*Dimostrazione:*

una chiave non presente nella tabella può essere collocata in uno qualsiasi degli  $m$  slot.

Una ricerca senza successo tocca tutte le chiavi nella lista corrispondente.

$$\text{Tempo di hashing (1)} + \text{lunghezza attesa lista} (\alpha) \Rightarrow \Theta(1 + \alpha)$$

Ricerca con successo

*Teorema:* in una tabella hash con concatenamento, una ricerca senza successo richiede un tempo atteso di  $\Theta(1 + \alpha/2)$ . Questa ricerca tocca in media metà delle chiavi nella lista corrispondente.

Il fattore di carico influenza quindi il costo computazionale delle operazioni:

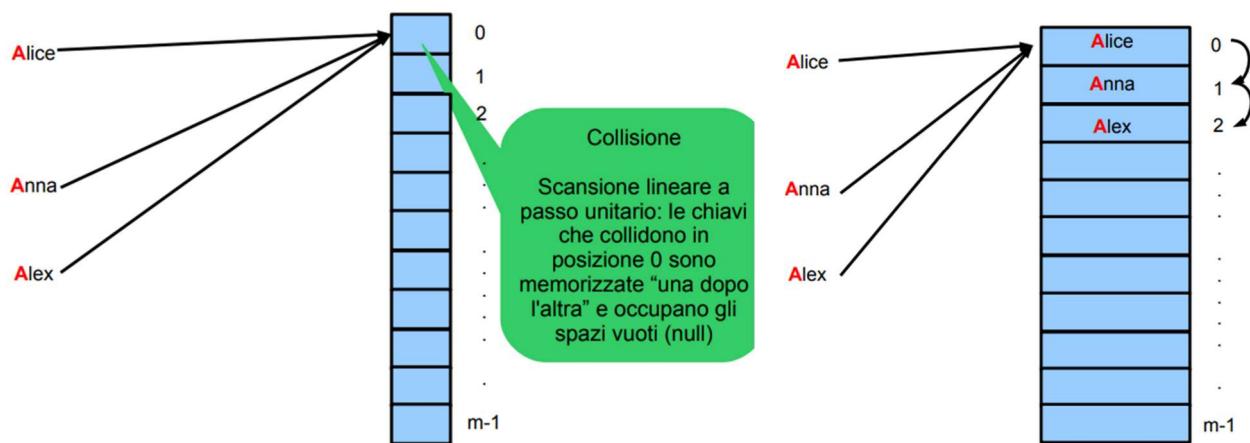
se  $n = O(m) \rightarrow \alpha = O(1)$  Quindi tutte le operazioni sono  $\Theta(1)$

### 6.4.2 Indirizzamento aperto (o scansione interna)

L'idea è quella di memorizzare tutte le chiavi nella tabella stessa: ogni slot contiene una chiave oppure *null*, se devo inserire un oggetto posso scegliere lo slot prescelto oppure, se è utilizzato, si cerca uno slot "alternativo".

La ricerca viene eseguita nello slot prescelto, e poi negli slot "alternativi" fino a quando non si trova la chiave oppure *null*.

Esempio:



Il fattore di carico in questo caso è compreso tra 0 e 1 (percentuale i quanto è " pieno"). La tabella può andare in *overflow* ed è quindi necessario usare tecniche di crescita e contrazione della tabella.

### 6.4.3 Ispezioni

L'ispezione è l'operazione che viene eseguita durante una ricerca di una chiave.

La sequenza di ispezione è la lista ordinata degli slot esaminati.

La funzione hash quindi viene estesa come

$$h: U \times [0, \dots, m - 1] \rightarrow [0, \dots, m - 1]$$

dove  $U$  è la chiave e  $[0, \dots, m - 1]$  è il tentativo di esaminare una cella.

L'output sarà una lista  $\{h(k, 0), h(k, 1), \dots, h(k, m - 1)\}$  che è una permutazione degli indici da 0 a  $m - 1$ . (può essere necessario esaminare ogni slot nella tabella e inoltre non vogliamo esaminare lo stesso slot più di una volta).

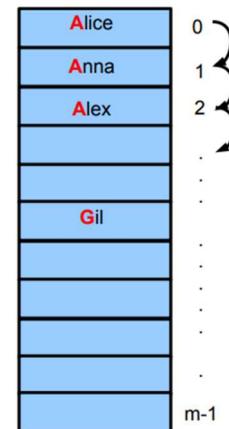
Alice: 0

Anna: 0, 1

Alex: 0, 1, 2

Gil: 5

Al: 0, 1, 2, 3



#### 6.4.3.1 Analisi del metodo di indirizzamento aperto

Nel caso pessimo, abbiamo un costo  $O(m)$  per tutte e tre le operazioni di base, dove  $m$  è la dimensione della tabella, questo perché nel caso pessimo ispezioniamo l'intera tabella.

Nel caso medio, il costo è influenzato dalla strategia di ispezione, questo anche sotto l'assunzione di hashing uniforme semplice.

Vediamo ora tre strategie di ispezione

#### 6.4.3.2 Ispezione lineare

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

Dove  $k$  è la chiave,  $i$  è il numero dell'ispezione e  $h'$  è una funzione hash ausiliaria.

Il primo elemento determina l'intera sequenza:

$$h'(k), h'(k) + 1, \dots, m_1, 0, 1, \dots, h'(k) - 1$$

Solo  $m$  sequenze di ispezione distinte sono possibili.

C'è il problema del *primary clustering*, cioè lunghe sotto-sequenze occupate che tendono a diventare più lunghe; uno slot vuoto preceduto da  $i$  slot pieni viene riempito con probabilità  $(i+1)/m$  e i tempi medi di inserimento e cancellazione crescono.

Esempio:

Inserimenti nel seguente ordine:

53, 75, 16, 73, 10, 33, 13, 76

|           |    |   |   |    |    |    |    |    |    |    |                                |
|-----------|----|---|---|----|----|----|----|----|----|----|--------------------------------|
| insert 53 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(53,0) = 3$                  |
|           | /  | / | / | 53 | /  | /  | /  | /  | /  | /  |                                |
| insert 75 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(75,0) = 5$                  |
|           | /  | / | / | 53 | /  | 75 | /  | /  | /  | /  |                                |
| insert 16 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(16,0) = 6$                  |
|           | /  | / | / | 53 | /  | 75 | 16 | /  | /  | /  |                                |
| insert 73 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(73,1) = 4 \quad h'(73) = 3$ |
|           | /  | / | / | 53 | 73 | 75 | 16 | /  | /  | /  |                                |
| insert 10 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(10,0) = 0$                  |
|           | 10 | / | / | 53 | 73 | 75 | 16 | /  | /  | /  |                                |
| insert 33 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(33,4) = 7 \quad h'(33) = 3$ |
|           | 10 | / | / | 53 | 73 | 75 | 16 | 33 | /  | /  |                                |
| insert 13 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(13,5) = 8 \quad h'(13) = 3$ |
|           | 10 | / | / | 53 | 73 | 75 | 16 | 33 | 13 | /  |                                |
| insert 76 | 0  | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9  | $h(76,3) = 9 \quad h'(76) = 6$ |
|           | 10 | / | / | 53 | 73 | 75 | 16 | 33 | 13 | 76 |                                |

#### 6.4.3.3 Ispezione quadratica

Funzione:  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$  con  $c_1 \neq c_2$

LA sequenza di ispezioni diventa quindi:

- L'ispezione iniziale è in  $h'(k)$
- Le ispezioni successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione  $i$
- Solo  $m$  sequenze di ispezione distinte sono possibili
- $c_1, c_2$  devono garantire una permutazione di  $[0, \dots, m - 1]$

C'è il problema del *clustering secondario*, cioè se due chiavi hanno la stessa ispezione iniziale, poi le loro sequenze sono identiche.

Esempio:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod 10$$

$$h'(k) = k \bmod 10, \quad c_1 = 0 \quad c_2 = 1$$

Inserimenti nello stesso ordine di prima:

53,75,16,73,10,33,13,76

|           |  |    |    |    |    |    |    |   |   |   |   |    |    |    |    |    |    |    |    |   |   |                       |
|-----------|--|----|----|----|----|----|----|---|---|---|---|----|----|----|----|----|----|----|----|---|---|-----------------------|
| insert 53 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>/</td><td>/</td><td>/</td><td>/</td><td>/</td><td>/</td></tr></table>        | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | /  | /  | /  | 53 | /  | /  | /  | /  | / | / | h(53,0) = 3           |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| /         | /  | /  | 53 | /  | /  | /  | /  | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| insert 75 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>/</td><td>75</td><td>/</td><td>/</td><td>/</td><td>/</td></tr></table>       | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | /  | /  | /  | 53 | /  | 75 | /  | /  | / | / | h(75,0) = 5           |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| /         | /  | /  | 53 | /  | 75 | /  | /  | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| insert 16 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>/</td><td>75</td><td>16</td><td>/</td><td>/</td><td>/</td></tr></table>      | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | /  | /  | /  | 53 | /  | 75 | 16 | /  | / | / | h(16,0) = 6           |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| /         | /  | /  | 53 | /  | 75 | 16 | /  | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| insert 73 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>73</td><td>75</td><td>16</td><td>/</td><td>/</td><td>/</td></tr></table>     | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | /  | /  | /  | 53 | 73 | 75 | 16 | /  | / | / | h(73,1) = 4 h(73) = 3 |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| /         | /  | /  | 53 | 73 | 75 | 16 | /  | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| insert 10 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>/</td><td>/</td><td>53</td><td>73</td><td>75</td><td>16</td><td>/</td><td>/</td><td>/</td></tr></table>    | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | 10 | /  | /  | 53 | 73 | 75 | 16 | /  | / | / | h(10,0) = 0           |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| 10        | /  | /  | 53 | 73 | 75 | 16 | /  | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| insert 33 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>/</td><td>/</td><td>53</td><td>73</td><td>75</td><td>16</td><td>33</td><td>/</td><td>/</td></tr></table>   | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | 10 | /  | /  | 53 | 73 | 75 | 16 | 33 | / | / | h(33,2) = 7 h(33) = 3 |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| 10        | /  | /  | 53 | 73 | 75 | 16 | 33 | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| insert 13 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>/</td><td>13</td><td>53</td><td>73</td><td>75</td><td>16</td><td>33</td><td>/</td><td>/</td></tr></table>  | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | 10 | /  | 13 | 53 | 73 | 75 | 16 | 33 | / | / | h(13,3) = 2 h(13) = 3 |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| 10        | /  | 13 | 53 | 73 | 75 | 16 | 33 | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| insert 76 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>76</td><td>13</td><td>53</td><td>73</td><td>75</td><td>16</td><td>33</td><td>/</td><td>/</td></tr></table> | 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | 10 | 76 | 13 | 53 | 73 | 75 | 16 | 33 | / | / | h(76,5) = 1 h(76) = 6 |
| 0         | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8 | 9 |   |   |    |    |    |    |    |    |    |    |   |   |                       |
| 10        | 76   | 13 | 53 | 73 | 75 | 16 | 33 | / | / |   |   |    |    |    |    |    |    |    |    |   |   |                       |

#### 6.4.3.4 Doppio hashing

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

$h_1$  e  $h_2$  sono due funzioni ausiliarie: la prima fornisce la prima ispezione e la seconda fornisce l'offset delle successive ispezioni.

Esempio:

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod 10$$

$$h_1(k) = k \bmod 10, \quad h_2(k) = (k \bmod 9) + 1$$

Inserimenti manco a dirtelo...

|           |  |   |    |    |    |    |    |    |   |   |   |    |    |   |    |    |    |    |    |    |   |                       |
|-----------|--|---|----|----|----|----|----|----|---|---|---|----|----|---|----|----|----|----|----|----|---|-----------------------|
| insert 53 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>/</td><td>/</td><td>/</td><td>/</td><td>/</td><td>/</td></tr></table>        | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | /  | /  | / | 53 | /  | /  | /  | /  | /  | / | h(53,0) = 3           |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| /         | /  | / | 53 | /  | /  | /  | /  | /  | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| insert 75 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>/</td><td>75</td><td>/</td><td>/</td><td>/</td><td>/</td></tr></table>       | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | /  | /  | / | 53 | /  | 75 | /  | /  | /  | / | h(75,0) = 5           |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| /         | /  | / | 53 | /  | 75 | /  | /  | /  | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| insert 16 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>/</td><td>75</td><td>16</td><td>/</td><td>/</td><td>/</td></tr></table>      | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | /  | /  | / | 53 | /  | 75 | 16 | /  | /  | / | h(16,0) = 6           |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| /         | /  | / | 53 | /  | 75 | 16 | /  | /  | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| insert 73 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>/</td><td>/</td><td>/</td><td>53</td><td>/</td><td>75</td><td>16</td><td>73</td><td>/</td><td>/</td></tr></table>     | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | /  | /  | / | 53 | /  | 75 | 16 | 73 | /  | / | h(73,2) = 7 h(73) = 3 |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| /         | /  | / | 53 | /  | 75 | 16 | 73 | /  | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| insert 10 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>/</td><td>/</td><td>53</td><td>/</td><td>75</td><td>16</td><td>73</td><td>/</td><td>/</td></tr></table>    | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | /  | / | 53 | /  | 75 | 16 | 73 | /  | / | h(10,0) = 0           |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| 10        | /  | / | 53 | /  | 75 | 16 | 73 | /  | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| insert 33 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>/</td><td>/</td><td>53</td><td>33</td><td>75</td><td>16</td><td>73</td><td>/</td><td>/</td></tr></table>   | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | /  | / | 53 | 33 | 75 | 16 | 73 | /  | / | h(33,3) = 4 h(33) = 7 |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| 10        | /  | / | 53 | 33 | 75 | 16 | 73 | /  | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| insert 13 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>/</td><td>/</td><td>53</td><td>33</td><td>75</td><td>16</td><td>73</td><td>13</td><td>/</td></tr></table>  | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | /  | / | 53 | 33 | 75 | 16 | 73 | 13 | / | h(13,1) = 8 h(13) = 5 |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| 10        | /  | / | 53 | 33 | 75 | 16 | 73 | 13 | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| insert 76 | <table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>10</td><td>76</td><td>/</td><td>53</td><td>33</td><td>75</td><td>16</td><td>73</td><td>13</td><td>/</td></tr></table> | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8 | 9 | 10 | 76 | / | 53 | 33 | 75 | 16 | 73 | 13 | / | h(76,1) = 1 h(76) = 6 |
| 0         | 1  | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 |   |   |    |    |   |    |    |    |    |    |    |   |                       |
| 10        | 76   | / | 53 | 33 | 75 | 16 | 73 | 13 | / |   |   |    |    |   |    |    |    |    |    |    |   |                       |

#### 6.4.4 Analisi dei costi di ispezione

| Esito ricerca      | Concatenamento       | Ispezione lineare                         | Ispezione quadratica<br>Hashing doppio    |
|--------------------|----------------------|---|---|
| Chiave trovata     | $\Theta(1 + \alpha)$ | $\frac{1}{2} + \frac{1}{2(1 - \alpha)}$   | $-\frac{1}{\alpha} \cdot \ln(1 - \alpha)$ |
| Chiave non trovata | $\Theta(1 + \alpha)$ | $\frac{1}{2} + \frac{1}{2(1 - \alpha)^2}$ | $\frac{1}{1 - \alpha}$                    |

Nel caso di gestione degli overflow mediante liste concatenate è possibile avere  $\alpha > 1$

Invece mediante indirizzamento aperto, è possibile avere  $\alpha \leq 1$ , ma una volta che l'array è pieno non è più possibile aggiungere altri elementi.

Commenti generali

Le prestazioni delle tabelle hash sono legate al fattore di carico  $\alpha$ , le collisioni sono molto probabili (vedi paradosso del compleanno), come strategia si può adottare quella di **mantenere il fattore di carico basso**; un fattore  $\alpha < 0.75$  è considerato ottimale.

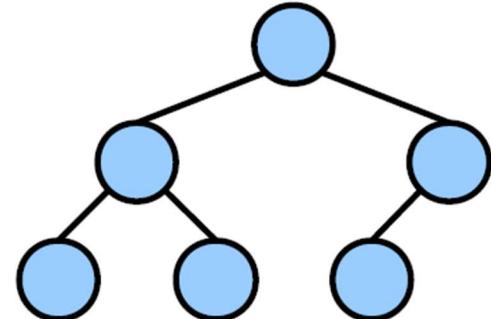
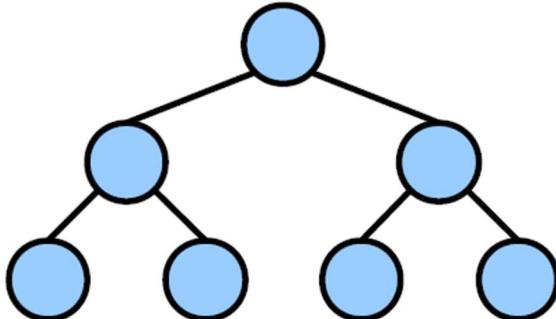
## 7 HEAP E SUE APPLICAZIONI

### 7.1 HEAP BINARI

Un albero binario perfetto è un albero in cui tutte le foglie hanno la stessa altezza  $h$  e i nodi interni hanno grado 2.

Un albero perfetto invece ha altezza  $h \approx \log N$ , dove  $N = \# \text{nodi} = 2^{h+1} - 1$

Un albero binario “quasi” perfetto è un albero perfetto fino al livello  $h - 1$ , dove tutti i nodi a livello  $h$  sono “compattati” a sinistra



#### 7.1.1 Alberi binari heap

Un albero binario quasi perfetto è un albero **max-heap** sse

- Ad ogni nodo viene associato un valore  $A[i]$
- $A[\text{Parent}(i)] \geq A[i]$  (i nodi “sopra” hanno valore *maggior*e dei loro figli)

Un albero binario quasi perfetto è un albero **min-heap** sse

- Ad ogni nodo viene associato un valore  $A[i]$
- $A[\text{Parent}(i)] \leq A[i]$  (i nodi “sopra” hanno valore *minore* dei loro figli)

#### 7.1.2 Array heap

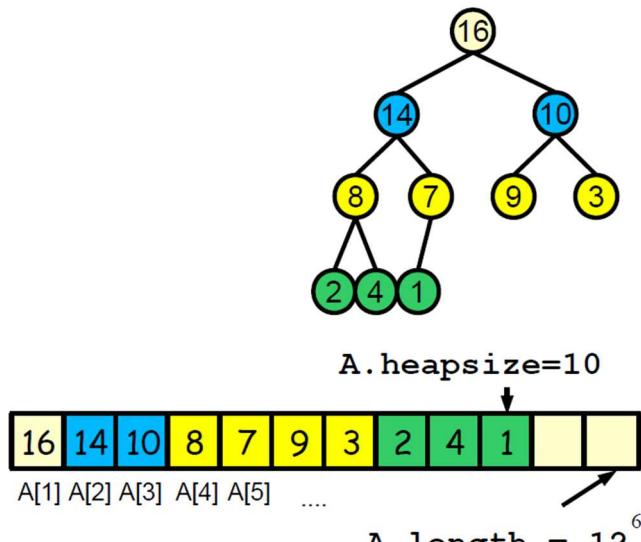
È la rappresentazione di un albero binario heap utilizzando un array

È organizzato così:

- ›  $A[1]$  contiene la radice
- ›  $\text{Parent}(i) = \text{Math.floor}(i/2)$
- ›  $\text{Left}(i) = 2 \cdot i$
- ›  $\text{Right}(i) = 2 \cdot i + 1$  (cella dopo  $\text{Left}(i)$ )

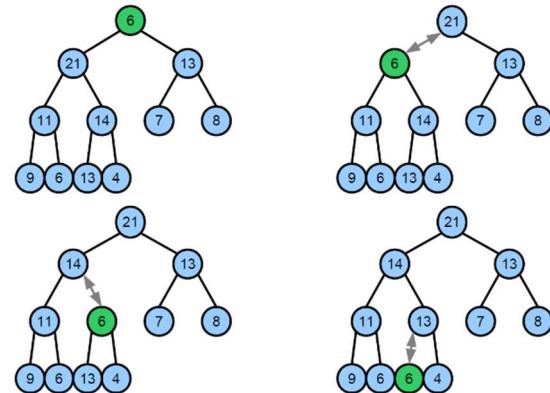
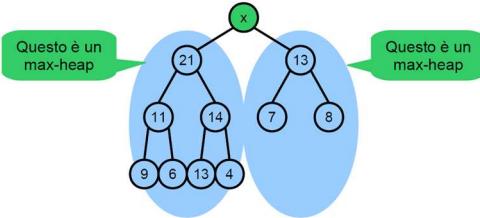
Le operazioni invece sono:

- # **findMax**: trova il valore massimo nell’heap  
costo:  $\Theta(1)$
- # **fixHeap**: ripristina la proprietà di max-heap  
Se rimpiazziamo  $A[1]$  con un valore qualsiasi, vogliamo che  $A[ ]$  torni ad essere un heap
- # **heapify**: Costruisce un heap a partire da un array caotico.
- # **deleteMax**: Rimuove l’elemento massimo da un max-heap  $A[ ]$



### 7.1.2.1 *fixHeap*

Immaginiamo di **dover rimpiazzare la radice**  $A[1]$  di un max-heap con un valore  $x$  qualsiasi



L'operazione fixHeap riordina l'albero in modo che sia ancora definibile max-heap;

Se  $x$  non è  $\geq$  dei figli, si può scambiare con il figlio con valore maggiore e procedere ricorsivamente

```

1. function fixHeap (int S[], int c, int i) {
2.   if (2*i>c) return;
3.   int max = 2*i;           // 50iglio sinistro
4.   if (2*i+1 <= c && S[2*i]<S[2*i+1])
5.     max = 2*i+1;          // figlio destro
6.   if (S[i] < S[max]) {
7.     int temp = S[max];    // risulta necessario fare lo
8.     S[max] = S[i];        // swap tra padre e figlio
9.     S[i] = temp;          // valore massimo
10.    fixHeap(S, c, max);  // si continua ricorsivamente
11.  }
12. }
```

### 7.1.2.2 *heapify*

Costruisce un heap a partire da un array

- $S[\dots]$  è un array; assumiamo che l'heap abbia  $n$  elementi
- $i$  è l'indice dell'elemento che diventerà la radice dell'heap
- $n$  indica l'indice dell'ultimo elemento dell'heap

```

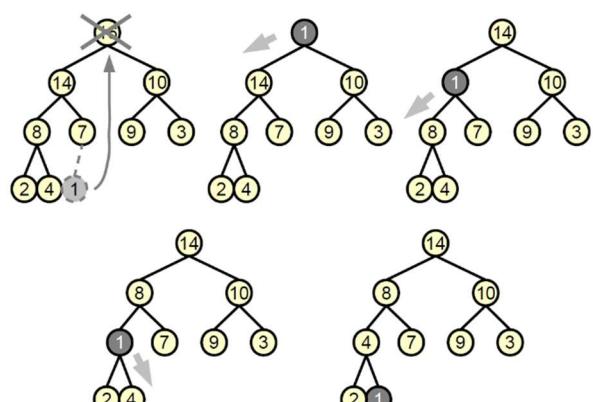
1. function heapify (int S[], int n; int i) {
2.   if (i > n) return;
3.   heapify(S, n, 2*i); // crea un heap radicato in S[2*i]
4.   heapify(S, n, 2*i+1); //      "      " radicato in S[2*i+1]
5.   heapify(S, n, i);
6. }
7. // per trasformare un array S in un heap:
8. // heapify(S, S.length, 1)
```

### 7.1.2.3 *deleteMax*

Vogliamo rimuovere la radice (ovvero il valore massimo) dall'heap, mantenendo la proprietà di max-heap

Al posto del vecchio valore  $A[1]$  metto il valore presente nell'ultima posizione dell'array heap

Applico poi fixHeap per ripristinare la proprietà di max-heap



### Riassunto costi computazionali heap

| Caso pessimo     |             |
|------------------|-------------|
| <b>fixHeap</b>   | $O(\log n)$ |
| <b>Heapify</b>   | $O(n)$      |
| <b>findMax</b>   | $O(1)$      |
| <b>deleteMax</b> | $O(\log n)$ |

### 7.1.3 Heapsort

Heapsort è una funzione che ordina l'array associato all'albero heap (min o max che sia)

Idea:

1. Costruire un max-heap a partire dal vettore  $A[\dots]$  originale, utilizzando heapify
2. Estrarre il massimo (findMax + deleteMax)
3. Inserire il massimo nell'ultima posizione di  $A[\dots]$
4. Ritornare al punto 2 finché l'heap diventa vuoto

```

1. function heapsort(int S[]) {
2.   heapify(S, S.length, 1);
3.   for (int c=S.length; c>1; c--) {
4.     int k = findMax(S);
5.     deleteMax(S, c);
6.     S[c] = k;
7.   }
8. }
```

➤ Costo computazionale:

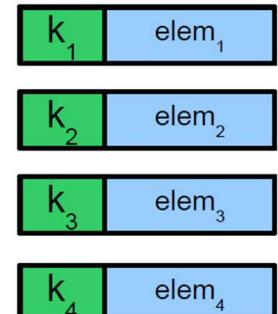
- $O(n)$  per heapify iniziale
- Ciascuna iterazione del ciclo 'for' costa  $O(\log c)$  (con  $c > n$ )

➤ Totale:

$$T(n) = O(n) + O\left(\sum_{c=n}^1 \log c\right) = \mathbf{O(n \log n)}$$

## 7.2 CODE CON PRIORITÀ

La coda con priorità è una struttura dati che mantiene il minimo (o il massimo) in un insieme dinamico di chiavi su cui è definita una relazione d'ordine totale, cioè un insieme di  $n$  elementi di tipo  $\text{elem}$  a cui sono associate chiavi.



### 7.2.1 Operazioni

- # **findMin() → elem**
  - restituisce un elemento associato alla chiave minima
- # **insert(elem e, chiave k)**
  - inserisce un nuovo elemento e con associata la chiave k
- # **delete(elem e)**
  - rimuove un elemento dalla coda (si assume di avere accesso diretto ad e)
- # **increaseKey(elem e, chiave c)**
  - Rimpiazza la chiave dell'elemento e con la nuova chiave c, se c è **maggior**e
- # **decreaseKey(elem e, chiave c)**
  - Rimpiazza la chiave dell'elemento e con la nuova chiave c, se c è **minore**

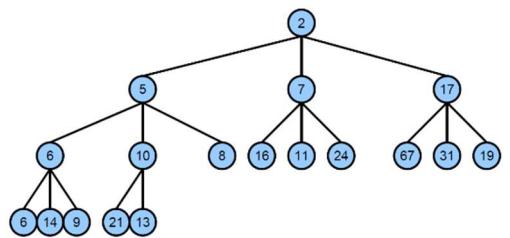
### 7.2.2 d-heap

Viene esteso "naturalmente" il concetto di min/max-heap binario.

Un heap binario è modellato su di un albero *binario*, invece il d-heap è modellato su un albero *d-ario*.

Ed ha le seguenti proprietà:

- un d-heap di altezza  $h$  è perfetto almeno fino alla profondità  $h - 1$   
le foglie al livello  $h$  sono accatastate a sinistra
- ogni nodo  $v$  contiene una chiave  $\text{chiave}(v)$  e un elemento  $\text{elem}(v)$ .  
Le chiavi appartengono ad un dominio totalmente ordinato
- ogni nodo diverso dalla radice ha chiave  $\geq$  a quella del padre



### 7.2.2.1 Altezza di un d-heap

Un d-heap con  $n$  nodi ha altezza  $O(\log_d n)$ :

Sia  $h$  l'altezza di un d-heap con  $n$  nodi. Il d-heap è perfetto fino al livello  $h - 1$ .

Un albero d-ario perfetto di altezza  $h - 1$  ha:

$$\sum_{i=0}^{h-1} d^i = \frac{d^h - 1}{d - 1}$$

nodi, quindi

$$\begin{aligned} \frac{d^h - 1}{d - 1} &< n \\ d^h &< n(d - 1) + 1 \\ h &< \log_d(n(d - 1) + 1) \Rightarrow O(\log_d n) \end{aligned}$$

### 7.2.2.2 Memorizzare un d-heap in un array

Come per un binary-heap, iniziamo dalla cella 1



il livello  $h$  inizia da

$$1 + \sum_{k=0}^{h-1} d^k = 1 + \frac{d^h - 1}{d - 1}$$

il livello  $h$  termina in

$$d^h + \sum_{k=0}^{h-1} d^k = d^h + \frac{d^h - 1}{d - 1}$$

Il padre di un nodo in posizione  $i$  è in posizione  $\lceil (i - 1)/d \rceil$

Per l'ultimo figlio non serve arrotondare, mentre per i fratelli precedenti bisogna arrotondare per eccesso)

### 7.2.2.3 Proprietà fondamentale dei d-heap

**La radice contiene un elemento con chiave minima**

La dimostrazione viene fatta per induzione sul numero di nodi:

Per  $n = 0$  (heap vuoto) oppure  $n = 1$  la proprietà vale

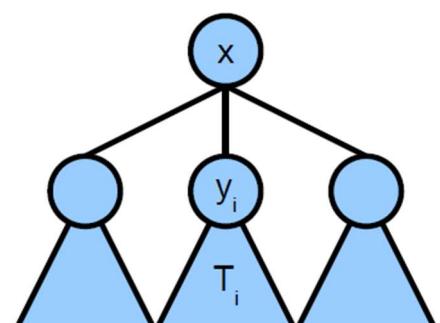
Supponiamo sia valida per ogni d-heap con al massimo  $n - 1$  nodi

Consideriamo un d-heap con  $n$  nodi; i sottoalberi radicati nei figli della radice sono a loro volta d-heap, con al più  $n - 1$  nodi.

La radice di  $T_i$  contiene il minimo di  $T_i$

La chiave radice  $x$  è  $\leq$  della chiave in ciascun figlio

Quindi la chiave in  $x$  è il minimo dell'intero heap



## 7.2.3 Implementazioni delle operazioni

### 7.2.3.1 Operazioni ausiliarie

```

1. procedura muoviAlto(v)
2.   while( v != root(T) and
chiave(v)<chiave(padre(v)) ) {
3.     // scambia di posto v e padre(v) in T;
4.     v := padre(v);
5.   endwhile

```

```

1. procedura muoviBasso(v)
2.   repeat forever
3.     if (v non ha figli) then
4.       return;
5.     else
6.       // u figlio di v con min chiave (u)
7.       if ( chiave(u) < chiave(v) ) then
8.         // scambia di posto u e v;
9.         v := u;
10.      else
11.        return;
12.      endif
13.    endif

```

### 7.2.3.2 *findMin()*

Restituisce l'elemento associato alla radice dell'heap (perché grazie alla proprietà sappiamo che un elemento con chiave minima è nella radice),

$$\Rightarrow \text{costo} = O(1)$$

### 7.2.3.3 *insert(elem e, chiave k)*

crea un nuovo nodo  $v$  con chiave  $k$  e valore  $e$ .

Aggiunge il nodo come ultima foglia a destra dell'ultimo livello (perché la proprietà di struttura sia soddisfatta). Per mantenere la proprietà di ordine, esegui  $\text{muoviAlto}(v)$  (che costa  $O(\log_d n)$  nel caso peggiore).

$$\Rightarrow \text{costo} = O(\log_d n)$$

### 7.2.3.4 *delete(elem e) e deleteMin()*

Sia  $v$  il nodo che contiene l'elemento  $e$  con chiave  $k$  (assumiamo di avere accesso diretto a  $v$ ).

Sia  $w$  l'ultima foglia a destra.

Setta  $\text{elem}(v) := \text{elem}(w)$

Setta  $\text{chiave}(v) := \text{chiave}(w)$

Stacca e cancella  $w$  dall'heap.

Esegui  $\text{muoviAlto}(v)$  (costo:  $O(\log_d n)$ ) e

Esegui  $\text{muoviBasso}(v)$  (costo:  $O(d \log_d n)$ )

*(Nota: una sola tra queste operazioni viene effettivamente eseguita; l'altra termina subito)*

$$\Rightarrow \text{costo} = O(d \log_d n)$$

### 7.2.3.5 *decreaseKey(elem e, chiave c)*

Sia  $v$  il nodo contenente  $e$  (assumiamo di avere accesso diretto a  $v$ )

Setta  $\text{chiave}(v) := \text{chiave}(v) - d$

Esegui  $\text{muoviAlto}(v)$

$$\Rightarrow \text{costo} = O(\log_d n)$$

### 7.2.3.6 *increaseKey(elem e, chiave c)*

analogo ma viene eseguito  $\text{muoviBasso}(v)$

$$\Rightarrow \text{costo} = O(\log_d n)$$

## 7.2.4 Costi per d-heap

| Funzioni                             | Costi computazionali |
|--------------------------------------|----------------------|
| <b>findMin() → elem</b>              | $O(1)$               |
| <b>insert(elem e, chiave k)</b>      | $O(\log_d n)$        |
| <b>delete(elem e)</b>                | $O(d \log_d n)$      |
| <b>deleteMin()</b>                   | $O(d \log_d n)$      |
| <b>increaseKey(elem e, chiave c)</b> | $O(d \log_d n)$      |
| <b>decreaseKey(elem e, chiave c)</b> | $O(\log_d n)$        |

## 8 UNION-FIND

Sono **strutture dati per insiemi disgiunti**. Immaginiamo di avere due insiemi disgiunti, ovvero contenenti elementi che appartengono solo ad o all'altro, matematicamente:  $U_1 \cap U_2 = \emptyset$

Le operazioni su questa struttura dati sono le basilari tra due insiemi:

- **makeSet(elem x)** crea un insieme singoletto a partire da un singolo elemento.
- **find(elem x) → set** cerca tra gli insiemi disponibili quello che contiene il dato elemento.
- **union(set x, set y)** unisce due insiemi tra di loro.

La struttura dati contiene un insieme dinamico di insiemi  $S = \{S_1, S_2, \dots, S_k\}$  tutti disgiunti tra loro. Tutti gli insiemi complessivamente contengono  $n \geq k$  elementi (dove  $n$  è la somma complessiva di tutti gli elementi,  $k$  è il numero degli insiemi; dato che ogni insieme è *almeno* un singoletto l'assunzione è ovvia). In più, ogni insieme è identificato da un **rappresentante univoco**.

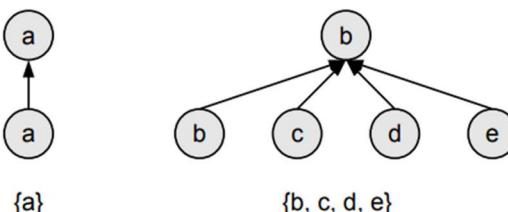
La scelta del rappresentante è importante, in quanto deve rispettare due leggi:

- › Il rappresentante di  $S_i$  deve essere un qualunque valore contenuto in  $S_i$
- › Chiamare **find** su  $k, K_1 \in S_i$  deve restituire lo stesso rappresentante
- › Il rappresentante può cambiare solo dopo una operazione di unione

Esempi di problemi risolvibili con una struttura *union find* sono quelli delle gestioni delle spedizioni o del tracciamento dei contatti su un PCB.

### 8.1 QUICKFIND

Ogni insieme viene rappresentato con un albero di altezza 1



Gli insiemi, quindi, sono strutturati in modo che le foglie siano gli elementi dell'insieme e il rappresentante è la radice.

# Le operazioni **makeSet()** e **find()** hanno tempo

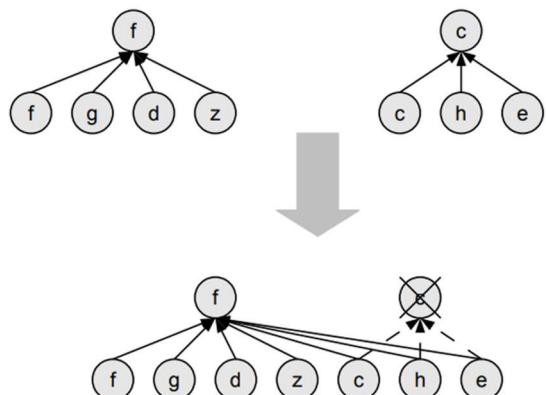
$O(1)$

- **makeSet(x)**: crea un albero in cui l'unica foglia è  $x$  e il rappresentante di  $x$  è  $x$  stesso
- **find(x)**: restituisce il puntatore al padre di  $x$

# L'operazione **union(A, B)** richiede più tempo

$O(n)$

- Tutte le foglie dall'albero  $B$  vengono spostate nell'albero  $A$ .
- Costo nel caso pessimo  $O(n)$ , essendo  $n$  il numero complessivo di elementi in entrambi gli insiemi disgiunti.



## 8.2 QuickUNION

L'implementazione del QuickUnion è basata sulla foresta; ogni insieme viene rappresentato come un albero radicato generico. Ogni nodo dell'albero contiene:

- L'oggetto
- Un puntatore al padre (la radice non ha il padre)
- Il rappresentante è la radice

Le operazioni sono:

### # makeSet(x)

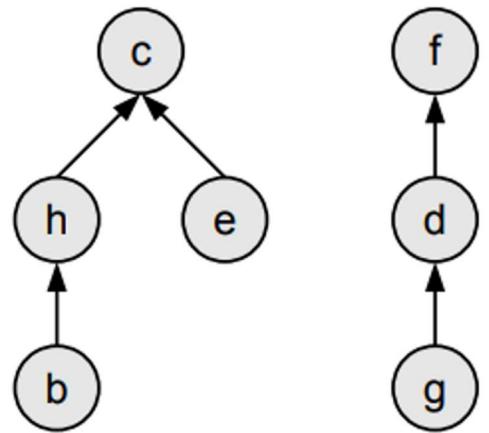
- Crea un albero con un unico nodo x
  - Costo:  $O(1)$  nel caso pessimo

### # find(x)

- Risale la lista dei padri di x fino a trovare la radice e restituisce la radice come oggetto rappresentante
  - Costo:  $O(n)$  nel caso pessimo

### # union(A, B)

- Appende l'albero B ad A, rendendo la radice di B figlia della radice di A
  - Costo:  $O(1)$  nel caso pessimo



{c, e, h, b}

{f, d, g}

## 8.3 RIEPILOGO

|           | QuickFind | QuickUnion |
|-----------|-----------|------------|
| makeSet() | $O(1)$    | $O(1)$     |
| union()   | $O(n)$    | $O(1)$     |
| find()    | $O(1)$    | $O(n)$     |

### 8.3.1 Considerazioni

Quando usare:

#### ➤ QuickFind?

Quando le `union()` sono rare e le `find()` sono frequenti

#### ➤ QuickUnion?

Quando le `find()` sono rare e le `union()` sono frequenti

## 8.4 EURISTICHE DI OTTIMIZZAZIONE

### 8.4.1 Euristiche su QuickFind

L'operazione `union()` è poco efficiente nel caso in cui uniamo a un singoletto un insieme grande, ma possiamo fare di meglio in questo caso. Invece che fare l'unione classica come abbiamo definito prima, possiamo guardare il "peso" di ogni albero (mantenuto come valore nella radice in tempo costante  $O(1)$  ad ogni operazione) e quando andiamo a fare l'unione svolgiamo il cambiamento dei campi "parent" sull'insieme che ha peso minore, ciò ci garantisce di eseguire meno operazioni nei casi pessimi.

Quando si sposta il primo insieme nel secondo la specifica scelta del rappresentante non viene rispettata; quindi, aggiorniamo la radice in modo tale che sia uguale a quella del primo albero.

Prendendo un insieme singoletto contenente  $x$ , esso potrà al massimo cambiare padre  $\log_2 n$  volte, poiché ogni volta che viene inserito in un altro insieme raddoppia la grandezza dell'insieme di appartenenza di  $x$ , e dunque il cambiamento sarà al massimo  $\log_2 n$  volte.

Ne segue che il costo nel caso pessimo è  $O(n/2) = \boxed{O(n)}$ , tuttavia nel caso medio facendo un costo ammortizzato si ha che il costo sarà  $O(\log_2 n)$ .

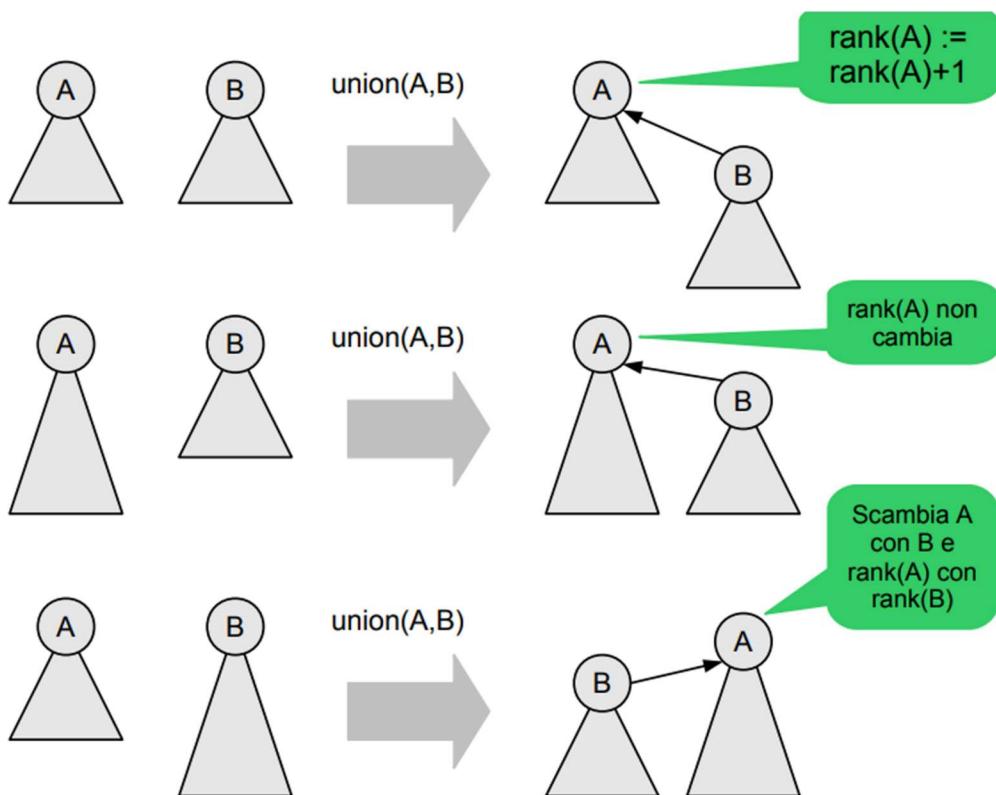
### 8.4.2 Euristiche su QuickUnion

L'operazione **find()** è particolarmente rallentata nel caso in cui l'albero assuma una profondità eccessiva, accomunando la sua struttura a quella di una lista.

Una euristica mirata a risolvere questo problema è quella che tiene traccia in ogni radice del **rank** (rango) di un albero, valore che ne indica la profondità massima.

Come nella euristica precedente grazie a questa metrica possiamo fare scelte intelligenti durante l'unione di due alberi. In particolare, quello che faremo sarà unire **y** in **x** solo se **y.rank < x.rank**, altrimenti faremo l'opposto ricordandoci di cambiare poi le radici per mantenere la proprietà del rappresentante.

In questo modo avremo alberi al massimo di profondità logaritmica. L'operazione **find()** che ha costo pessimo pari alla profondità massima dell'albero avrà dunque un costo massimo  $O(\log_2 n)$



|                  | QuickFind | QuickUnion | QuickFind<br>eur. Peso | QuickUnion<br>eur. Rank |
|------------------|-----------|------------|------------------------|-------------------------|
| <b>makeSet()</b> | $O(1)$    | $O(1)$     | $O(1)$                 | $O(1)$                  |
| <b>union()</b>   | $O(n)$    | $O(1)$     | $O(\log n)$            | $O(1)$                  |
| <b>find()</b>    | $O(1)$    | $O(n)$     | $O(1)$                 | $O(\log n)$             |

## 9 TECNICHE ALGORITMICHE

### # Divide-et-impera

- Un problema viene diviso in sotto-problemi, che vengono risolti ricorsivamente

### # Algoritmi greedy

- Ad ogni passo si fa sempre la scelta che in quel momento appare ottima, le scelte fatte non vengono mai disfatte

### # Programmazione dinamica

- La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi

### 9.1 DIVIDE-ET-IMPERA

Un problema viene suddiviso in sotto-problemi che vengono risolti ricorsivamente (top-down)

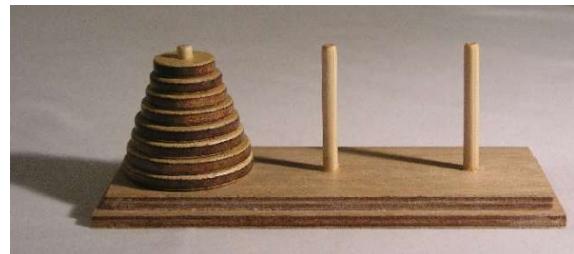
Negli algoritmi di questo tipo quindi si possono trovare queste “fasi” che però sono diverse a seconda di quello che si vuole fare (per esempio, nel QuickSort non c’è la fase “*combina*”)

- *Divide*: dividi il problema in sotto-problemi indipendenti di dimensioni minori
- *Impera*: risovi i sotto-problemi ricorsivamente
- *Combina*: unisci le soluzioni dei sotto-problemi per costruire la soluzione del problema di partenza

#### 9.1.1 Le torri di Hanoi

Le torri di Hanoi è un gioco matematico che useremo per provare ad applicare questa tecnica algoritmica.

Ci sono tre pioli e ci sono  $n$  dischi messi a pila su un piolo, i dischi hanno diametro decrescente verso l’alto. Lo scopo del gioco è spostare la pila dal primo al terzo piolo, muovendo solo un disco alla volta e senza mai impilare un disco più grande su uno più piccolo.



##### 9.1.1.1 Soluzione divide-et-impera

```

1. Hanoi (Stack p1, Stack p2, Stack p3, int n)
2.   if (n = 1) then
3.     p3.push(p1.pop())
4.   else
5.     Hanoi(p1, p3, p2, n-1)
6.     p3.push(p1.pop())
7.     Hanoi(p2, p1, p3, n-1)
8.   endif
  
```

##### Divide:

|     |           |         |
|-----|-----------|---------|
| n-1 | dischi da | p1 a p2 |
| 1   | disco da  | p1 a p3 |
| n-1 | dischi da | p2 a p3 |

##### Impera:

Esegui ricorsivamente gli spostamenti

##### 9.1.1.2 Costo computazionale

- $T(1) = 1$
- $T(n) = 2 T(n - 1) + 1$  per  $n > 1$

Quindi nel caso  $n$  si continua a fare chiamate per  $n - 1$  e moltiplicando continuamente 2 finché non arriva al caso base (analisi con la tecnica dell’iterazione)

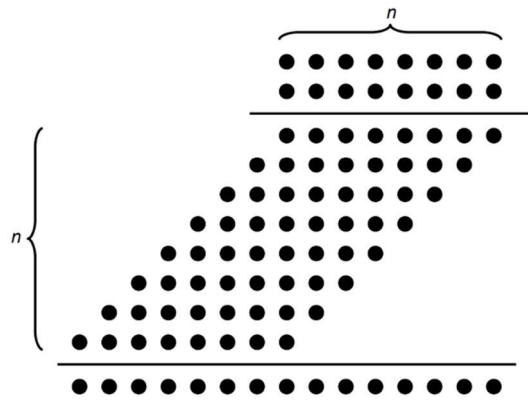
Quindi il costo della funzione è  $O(2^n)$

### 9.1.2 Moltiplicazione di interi

Consideriamo due interi  $X$  e  $Y$  di  $n$  cifre decimali:

$$X = x_{n-1}x_{n-2} \cdots x_1x_0 = \sum_{i=0}^{n-1} x_i \times 10^i$$

$$Y = y_{n-1}y_{n-2} \cdots y_1y_0 = \sum_{i=0}^{n-1} y_i \times 10^i$$



Calcolando il prodotto con l'algoritmo della moltiplicazione in colonna (quello insegnato alle elementari) il costo è  $O(n^2)$ . Noi proveremo a fare di meglio approcciando una tecnica divide-et-impera.

#### 9.1.2.1 Miglioramento

Supponiamo che  $X$  e  $Y$  abbiano lo stesso numero di cifre (in caso contrario si aggiungono degli 0 all'inizio). Dividiamo le sequenze di cifre in due parti uguali:

$$X = X_1 \times 10^{n/2} + X_0$$

|       |       |
|-------|-------|
|       |       |
| $X_1$ | $X_0$ |

$$Y = Y_1 \times 10^{n/2} + Y_0$$

|       |       |
|-------|-------|
|       |       |
| $Y_1$ | $Y_0$ |

Quindi si può calcolare il prodotto come:

$$\begin{aligned} X \times Y &= (X_1 \cdot 10^{n/2} + X_0) \times (Y_1 \cdot 10^{n/2} + Y_0) \\ &= (X_1 \cdot Y_1) \times 10^n + (X_1 Y_0 + X_0 Y_1) \times 10^{n/2} + X_0 Y_0 \end{aligned}$$

Quindi per risolvere la moltiplicazione posso ridurmi a fare la somma tra 4 prodotti di numeri a  $n/2$  cifre.

La moltiplicazione per  $10^n$  richiede tempo  $O(n)$  (che equivale ad uno shift a sinistra di  $n$  posizioni).

Possiamo quindi scrivere la relazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 4T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

Soluzione (usando MT, caso 1):  $O(n^2)$ , quindi per ora non abbiamo migliorato nulla

Ma se poniamo

$$\begin{aligned} P_1 &= (X_1 + X_0) \times (Y_1 + Y_0) \\ P_2 &= X_1 Y_1 \\ P_3 &= X_0 Y_0 \end{aligned}$$

allora possiamo scrivere

$$X \times Y = P_2 \cdot 10^n + (P_1 - P_2 - P_3) \times 10^{n/2} + P_3$$

Il calcolo di  $P_1$ ,  $P_2$  e  $P_3$  richiede in tutto solo 3 prodotti tra numeri di  $n/2$  cifre, invece delle 4 di prima.

Analizzando l'algoritmo quindi vediamo che è meglio fare più somme rispetto a delle moltiplicazioni.

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ 3T(n/2) + c_2 n & \text{altrimenti} \end{cases}$$

Usando MT, caso 1:  $O(n^{\log_2 3}) \approx O(n^{1.59})$

### 9.1.3 Sottovettore non vuoto di valore massimo

Prendiamo un vettore  $v[1 \dots n]$  di  $n$  valori arbitrari. Vogliamo individuare un sottovettore non vuoto di  $v$  la cui somma degli elementi sia massima

|   |    |    |   |    |   |   |    |   |    |    |
|---|----|----|---|----|---|---|----|---|----|----|
| 3 | -5 | 10 | 2 | -3 | 1 | 4 | -8 | 7 | -6 | -1 |
|---|----|----|---|----|---|---|----|---|----|----|

?

Quanti sono i sottovettori di  $v$ ?

- › 1 sottovettore di lunghezza  $n$
- › 2 sottovettori di lunghezza  $n - 1$
- › ...
- ›  $k$  sottovettori di lunghezza  $n - k + 1$
- › ...
- ›  $n$  sottovettori di lunghezza 1

Per cui avremo un numero di vettori pari a  $O(n(n - 1)/2) = O(n^2)$

#### 9.1.3.1 Soluzione 1 (forza bruta)

Questo algoritmo non ha nessuna idea specifica: controlliamo la somma di TUTTI i sottovettori e cerco il più grande.

La complessità sarà relativa al numero di sottovettori e alla somma di tutti gli elementi, quindi  $O(n^3)$

```

1. double sommaMax1 (double v[n])
2.   double smax = v[1];
3.   for (int i = 1; i < n; i++) {
4.     for (int j = 1; j < n; j++) {
5.       double s = 0;
6.       for (int k = i; k < j; k++) {
7.         s = s + v[k];
8.       }
9.       if (s > smax) {
10.         smax = s;
11.     }
12.   }
13. }
14. return smax;
```

#### 9.1.3.2 Soluzione 2 (forza bruta-migliorato)

Un modo per ottimizzare l'algoritmo di prima è partire dal primo valore e calcolare i sotto-valori che partono dal primo, poi dal secondo e così via, risparmiando così di risommare tutti i valori precedenti.

Il costo è quindi  $O(n^2)$ , perché nell'algoritmo ci siamo risparmiati un for.

```

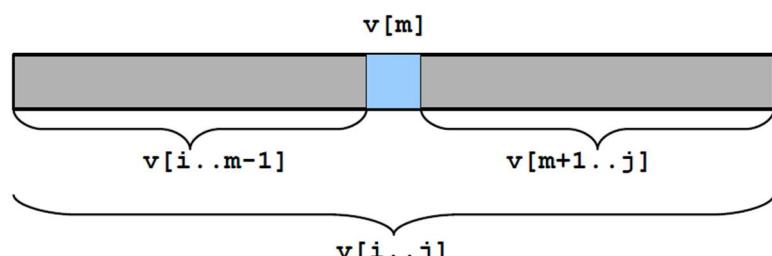
1. double sommaMax2 (double v[n])
2.   double smax = v[1];
3.   for (int i = 1; i < n; i++) {
4.     double s = 0;
5.     for (int j = i; j < n; j++) {
6.       s = s + v[j];
7.       if (s > smax) {
8.         smax = s;
9.       }
10.    }
11.  }
12. return smax;
```

#### 9.1.3.3 Soluzione 3 (divide-et-impera)

L'idea è di dividere il vettore in due parti separate dall'elemento centrale  $v[m]$ .

Il sottovettore che cerchiamo potrebbe quindi trovarsi:

- Nella prima metà  $v[0, \dots, m - 1]$
- Nella seconda metà  $v[m + 1, \dots, j]$
- "a cavallo" tra la prima e la seconda

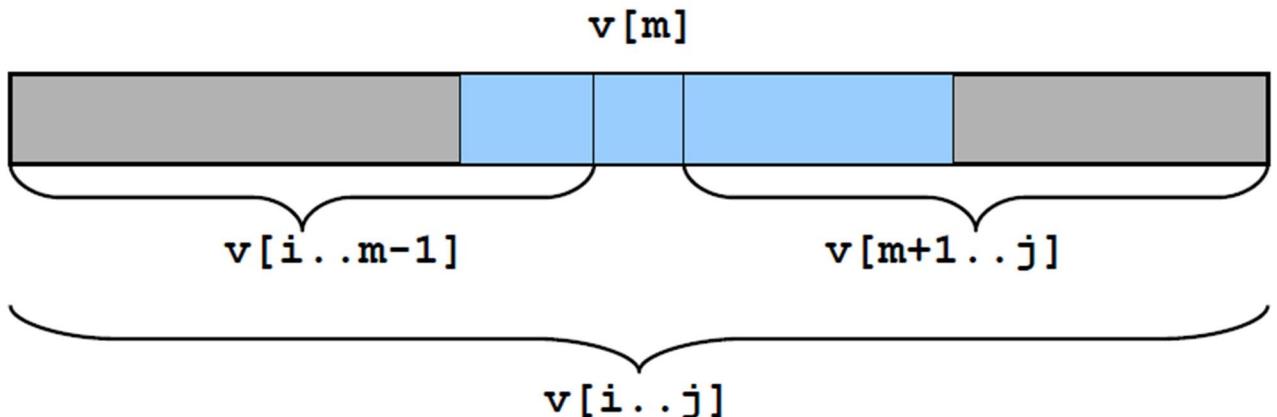


Quindi la parte complicata è ricercare il sottovettore di somma massima che contiene  $v[m]$

Per trovare cercherò:

- La somma massima  $Sa$  tra tutti i sottovettori che hanno come ultimo valore  $v[m - 1]$
- La somma massima  $Sb$  tra tutti i sottovettori che hanno come primo valore  $v[m + 1]$

Quindi il sottovettore di somma massima che include  $v[m]$  sarà  $Sa + Sb + v[m]$ .



```

1. real SommaMaxDI (real V[n], int i, int j)
2.   if (i > j) return 0;
3.   else if (i == j) return V[i];
4.   else {
5.     m = floor((i + j) / 2);
6.     real l = SommaMaxDI(V, i, m-1);
7.     real r = SommaMaxDI(V, m+1, j);
8.     real sa = 0, sb = 0, s = 0;
9.     int k;
10.    for (k = m-1; k >= i; k--) {
11.      s = s + V[k];
12.      if (s > sa) sa = s;
13.    }
14.    s = 0;
15.    for (k = m+1; k <= j; k++) {
16.      s = s + V[k];
17.      if (s > sb) sb = s;
18.    }
19.    Return max(l, r, V[m]+sa+sb);
20.  }

```

Il costo computazionale della funzione sarà relativo al numero di chiamate ricorsive e al fatto che si scorre l'array con un for per ogni chiamata ricorsiva, ma vediamo la relazione di ricorrenza:

$$T(n) = 2T(n/2) + n$$

$2T(n/2)$  perché vengono eseguite due chiamate ricorsive su metà dei valori del vettore.

Quindi, usando il Master Theorem (caso 2, perché  $\alpha = \beta$ ) il costo sarà

$$O(n \log n)$$

## 9.2 ALGORITMI GREEDY

Per rendere l'algoritmo più veloce possiamo usare la tecnica greedy quando:

- tra le molte scelte *se ne può identificare una migliore che sicuramente ci poterà alla soluzione*
- quando l'algoritmo ha una *struttura ottima* e dunque siamo sicuri che fatta tale scelta la struttura del problema non varia.

Non tutti gli algoritmi offrono scelte greedy e non sempre questa soluzione è conveniente, ma ci sono problemi interessanti da analizzare.

### 9.2.1 Problema del resto

In input ci sarà un numero intero positivo  $R$  che rappresenta un importo (in centesimi di euro) da erogare

In output dovremo avere il minimo numero intero di monete necessarie per erogare il resto di  $R$  centesimi usando solo monete da 50c, 20c, 10c, 5c, 2c, 1c. Disponiamo di infinite monete di ogni taglio.

#### SOLUZIONE

La soluzione ottimale è quella di partire dal taglio maggiore di moneta e vedere se possiamo usarla per il resto. Se sì, incremento il numero di monete usate e tolgo il taglio di monete usato e passo alla prossima moneta.

```

1. int RestoGreedy(int R, int T[n]) {           // R = resto da erogare
2.     ordina.decrecente(T);                     // T[n] = gli n tagli di monete a disposizione
3.     int nm = 0;                             // output = numero totale di monete da erogare
4.     int i = 1;
5.     while (R > 0 && i <= n) {
6.         if (R >= T[i]) {
7.             R = R - T[i];
8.             nm = nm + 1;
9.         } else {
10.            i = i + 1;
11.        }
12.    }
13.    if(R > 0) {
14.        error: "resto non erogabile"
15.    } else {
16.        return nm;
17.    }
18. }
```

Questo algoritmo è generalizzato a prendere in input anche monete con tagli diversi dall'esempio e quindi si usa un ciclo per controllare iterativamente quante monetine usare.

Il costo computazionale, quindi, sarà di  $O(n \log n)$  a causa dell'ordinamento decrescente delle monetine.

#### Osservazioni:

La tecnica greedy in questo caso funziona perché abbiamo preso in esame un *sistema monetario canonico*. Infatti, in sistemi monetari non canonici questo algoritmo potrebbe non dare la soluzione corretta oppure potrebbe non dare proprio una soluzione.

Un esempio può essere erogare 6 centesimi di resto con monete da 4, 3, 1 (sol greedy 4+1+1, ottimale 3+3)

Invece, immaginiamo che venga abolita la moneta da un centesimo. Se devo quindi erogare 6 centesimi di resto, l'algoritmo scritto sopra, prende 5 centesimi e poi non riesce più a trovare la soluzione perché può prendere solo la monetina da 2 centesimi, quando la soluzione ottimale sarebbe 2+2+2 = 3 monete.

Vedremo che con la programmazione dinamica questo problema verrà risolto e darà una soluzione corretta con ogni sistema monetario.

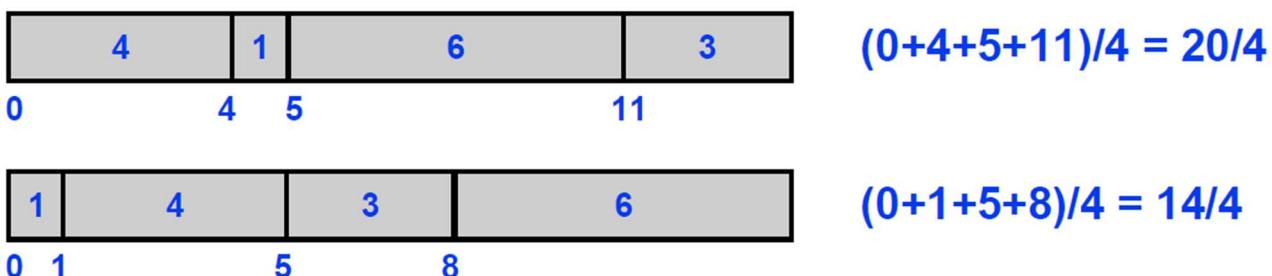
### 9.2.2 Problema di scheduling

Lo scheduler:

è un componente software del kernel di un sistema operativo che si occupa di assegnare la CPU a un determinato processo.

Un processo è un *job* che arriva a un tempo  $t$  e impegna  $n$  tempi (sec, msec, ...) per essere completato. Il modo con cui lo scheduler decide a chi assegnare la CPU è definito da un algoritmo

Definito quindi il contesto, vogliamo che il nostro scheduler esegua i processi *minimizzando il tempo medio di completamento*.



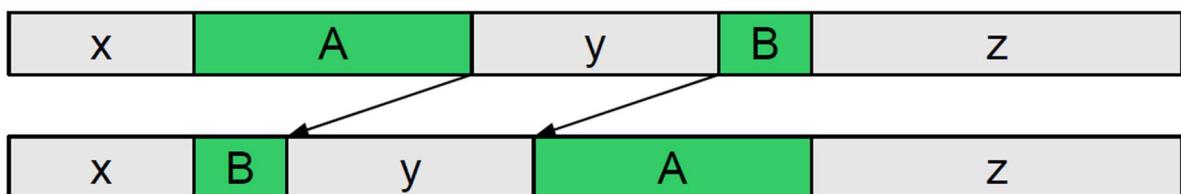
**SOLUZIONE:**

Il tempo totale sarà lo stesso, ma noi andiamo a sommare i tempi di completamento dei vari processi.

Quindi la soluzione ottimale sarà quella di eseguire i processi in ordine crescente di tempo perché vado a minimizzare la somma dei tempi finali.

Dimostrazione: (per assurdo)

Cerchiamo di arrivare ad un assurdo, considerando un ordinamento dei job in cui un job “lungo”  $A$  viene schedulato prima di uno “corto”  $B$  è quello migliore.  $x, y$  e  $z$  sono sequenze di altri job.



Se scambio  $A$  e  $B$  noto che tutte le terminazioni dei processi tra  $B$  e  $A$  hanno tempi medi di esecuzione migliore. Quindi non sarà sicuramente il migliore. Quindi è un assurdo.

### COSTO COMPUTAZIONALE

Il costo di questa soluzione, quindi, sarà relativa a un semplice ordinamento dei processi e alla somma. Quindi il costo sarà  $O(n \log n)$

### 9.2.3 Problema della compressione (codifica di Huffman)

Vogliamo trovare la quantità di bit necessaria per rappresentare una sequenza binaria in modo che essa sia la minore possibile, ovvero che il file sia il più compresso possibile. Abbiamo alcuni vincoli:

- Useremo la *funzione di codifica* che prende un carattere dell’alfabeto e ci restituisce una sequenza di caratteri per rappresentarlo.
- Presa una sequenza di caratteri  $c_1, c_2, \dots, c_n$  verrà codificato come  $f(c_1), f(c_2), \dots, f(c_n)$
- Una volta che una sequenza è stata codificata dev’essere *sempre possibile decodificarla* tramite una lettura *sequenziale* (bit-after-bit) di tale codifica.

Problema: data la sequenza  $c_1, c_2, \dots, c_n$  definire una funzione di codifica  $f()$  che minimizza la lunghezza della codifica  $f(c_1), f(c_2), \dots, f(c_n)$

### **9.2.3.1 Codici a lunghezza fissa**

Il modo classico per codificare dei caratteri è usare una lunghezza fissa di bit per ogni carattere:

Supponiamo di avere un file di  $n$  caratteri

- Possibili caratteri: 'a'      'b'      'c'      'd'      'e'      'f'
  - Frequenze:                  45%    13%    12%    16%    9%    5%

Codifica tramite ASCII (8 bit per carattere)

- Dimensione totale:  $8n$  bit

Codifica basata sull'alfabeto (3 bit per carattere)

- Codifica: 000 001 010 011 100 101
  - Dimensione totale  $3n$  bit

### **9.2.3.2 Codici a lunghezza variabile**

Nella codifica a lunghezza variabile assegno a ogni lettera una codifica di bit con lunghezza non prefissata. Alla lettera più frequente verrà assegnata una codifica più corta per massimizzare il guadagno.

Il costo viene trovato sommando il prodotto delle frequenze delle lettere con il numero di bit usati per la codifica.

- |                 |     |     |     |     |      |      |
|-----------------|-----|-----|-----|-----|------|------|
| - Caratteri:    | 'a' | 'b' | 'c' | 'd' | 'e'  | 'f'  |
| - Frequenze:    | 45% | 13% | 12% | 16% | 9%   | 5%   |
| - Codifica      | 0   | 101 | 100 | 111 | 1101 | 1100 |
| - Costo totale: |     |     |     |     |      |      |

$$(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4) \cdot n = 2.24n$$

### *Codice “a prefisso”*

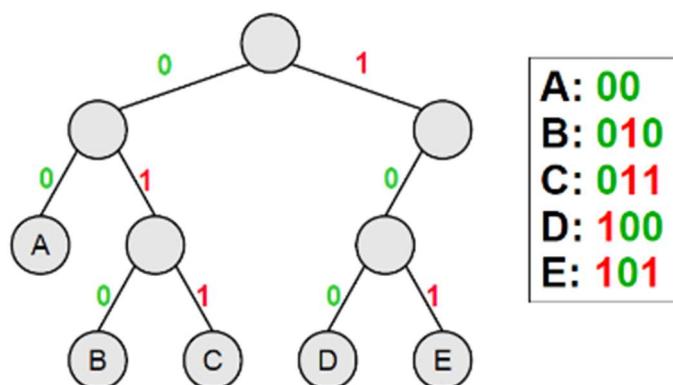
Nessun codice è un prefisso di un altro codice, condizione necessaria per permettere sempre la decodifica.

### 9.2.3.3 Codifica di Huffman

La codifica di Huffman risolve il problema della compressione. Rappresentiamo il codice con un albero binario:

- Figlio sinistro: 0      Figlio destro: 1
  - Caratteri dell'alfabeto sulle foglie

La proprietà del prefisso deve valere quindi solo le foglie dell'albero contengono le lettere dell'alfabeto.



## ALGORITMO DI HUFFMAN

Lo scopo dell'algoritmo è minimizzare la lunghezza dei caratteri che compaiono più frequentemente. Intuitivamente, le foglie più alte saranno assegnate a caratteri con più frequenza.

**Costruiamo il codice:**

Passo 1: Costruiamo una lista ordinata di nodi, in cui ogni nodo contiene un carattere e il numero di volte in cui quel carattere compare nel file.

**Passo 2:** Rimuovere i due nodi con frequenze minori

Passo 3: Collegarli a un nodo padre etichettato con la frequenza combinata

Passo 4: Aggiungere il nodo combinato alla lista, mantenendola ordinata in base alle frequenze

Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista.

Al termine si etichettano gli archi dell'albero con 0/

```

1. Tree Huffman(real f[n], char c[n]) {
2.   Q = new MinPriorityQueue();
3.   int i;
4.   for(i = 1; i < n; i++) {
5.     z = new TreeNode(f[i], c[i]);
6.     Q.insert(f[i], z);
7.   }
8.   for(i = 1; i < n-1; i++) {
9.     z1 = Q.findMin();    Q.deleteMin();
10.    z2 = Q.findMin();   Q.deleteMin();
11.    z = new TreeNode(z1.f + z2.f, ',');
12.    z.left = z1;
13.    z.right = z2;
14.    Q.insert(z1.f + z2.f, z);
15.  }
16.  return Q.findMin();
17. }
```

Il primo ciclo ha costo  $n \log_2 n$ , mentre il secondo  $n \log_2 n$

Costo:  $\Theta(n \log_2 n + n \log_2 3n) = O(n \log_2 n)$

#### 9.2.4 Sunto

Vantaggi:

- Semplici da programmare
- Solitamente efficienti
- Quando è possibile la proprietà di scelta greedy danno la soluzione ottima
- La soluzione sub-ottima può essere accettabile

Svantaggi:

- Non tutti i problemi ammettono una soluzione greedy
- In certi casi gli algoritmi greedy non danno la soluzione ottima, ma possono essere comunque applicati se ci si accontenta di una soluzione non necessariamente ottima.

### 9.3 PROGRAMMAZIONE DINAMICA

La programmazione dinamica è una tecnica usata per trovare la soluzione a problemi che richiedono la *soluzione ottima*. Questa tecnica algoritmica si basa sul costruire la soluzione a partire da un insieme di sottoproblemi ripetuti. Ogni volta che troviamo una soluzione di un problema che appare più volte, lo si va a salvare in una struttura dati per risolvere problemi che lo hanno come sottoproblema.

La sottostruttura deve essere ottima, ovvero dev'essere possibile combinare soluzioni dei sottoproblemi per trovare la soluzione di un problema più “grande”.

Divide-et-impera:

- Tecnica ricorsiva
- Approccio *top-down*
- Vantaggiosa quando i sottoproblemi sono indipendenti

Programmazione dinamica:

- Tecnica iterativa
- Approccio *bottom-up*
- Vantaggiosa quando ci sono sottoproblemi ripetuti
- Si usa una struttura dati di appoggio

#### ?) QUANDO APPLICARE LA PROGRAMMAZIONE DINAMICA?

- **Sottostruttura ottima**
  - Deve essere possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più “grande”
- **Sottoproblemi ripetuti**
  - Un sottoproblema compare più volte

### 9.3.1 Sottovettore di valore massimo

È lo stesso problema definito e risolto con divide-et-impera (costo pseudo-lineare  $O(n \log n)$ )

La soluzione proposta con la tecnica di programmazione dinamica è la più efficiente che si possa trovare.

#### IDEA

Sia  $P(i)$  il problema che consiste nel determinare il valore massimo della somma degli elementi dei sottovettori non vuoti del vettore  $V[1, \dots, i]$  che hanno  $V[i]$  come ultimo elemento.

Sia  $S[i]$  il valore della soluzione di  $P[i]$

$S[i]$  è la massima somma degli elementi dei sottovettori di  $V[1..i]$  che hanno  $V[i]$  come ultimo elemento

La soluzione  $S$  al problema può essere espressa come

$$S = \max_{1 \leq i \leq n} S[i]$$

$P(1)$  ammetta un'unica soluzione

$$S[1] = V[1]$$

Consideriamo il problema generico  $P(i), i > 1$

Supponiamo di avere già risolto il problema  $P(i - 1)$ , e quindi di conoscere  $S[i - 1]$

- # Se  $S[i - 1] + V[i] \geq V[i] \Rightarrow S[i] = S[i - 1] + V[i]$  se aumenta, ho trovato un sv maggiore
- # Se  $S[i - 1] + V[i] < V[i] \Rightarrow S[i] = V[i]$  se non aumenta non cambia

## Esempio

|     |   |    |    |   |    |   |   |    |   |    |    |
|-----|---|----|----|---|----|---|---|----|---|----|----|
| V[] | 3 | -5 | 10 | 2 | -3 | 1 | 4 | -8 | 7 | -6 | -1 |
|-----|---|----|----|---|----|---|---|----|---|----|----|

|     |   |    |    |    |   |    |    |   |    |   |   |
|-----|---|----|----|----|---|----|----|---|----|---|---|
| S[] | 3 | -2 | 10 | 12 | 9 | 10 | 14 | 6 | 13 | 7 | 6 |
|-----|---|----|----|----|---|----|----|---|----|---|---|

$$S[i] = \max\{V[i], V[i] + S[i - 1]\}$$

```

1. real sottovettoreMax(real V[n]) {
2.   real S[n];
3.   S[1] = V[1];
4.   int imax = 1;
5.   for (int i = 2; i < n; i++) {
6.     if (S[i-1]+V[i] >= V[i]) {
7.       S[i] = S[i-1]+V[i];
8.     } else {
9.       S[i] = V[i];
10.    }
11.    if (S[i] > S[imax]) {
12.      imax = i;
13.    }
14.  }
15.  return S[imax];
16. }
```

```

1. int indiceInizio(real V[n], real S[n], int imax) {
2.   int i = imax;
3.   while (S[i] != V[i]) {
4.     i = i - 1;
5.   }
6. }
```

### 9.3.2 Problema dello Zaino (Knapsack problem)

- Abbiamo un insieme  $X[1..n]$  di oggetti.
- L'oggetto  $i$ -esimo ha peso  $p[i]$  e valore  $v[i]$ .
- Disponiamo di uno zaino in grado di trasportare un peso massimo  $P$ .

Vogliamo determinare un sottoinsieme  $Y \subseteq X$  tale che:

- › il peso complessivo degli oggetti sia  $\leq P$
- › il valore complessivo degli oggetti sia il massimo

#### Soluzione greedy #1

Ad ogni passo, scelgo l'oggetto di valore massimo che è possibile mettere nello zaino.

Questo algoritmo non fornisce sempre la soluzione ottima

#### Soluzione greedy #2

Definiamo un valore specifico per ogni oggetto =  $v[i]/p[i]$

Ad ogni passo, scelgo l'oggetto con valore specifico massimo che è possibile mettere nello zaino.

Anche questa soluzione non da sempre un risultato ottimale.

#### Soluzione dinamica

- Definizione dei sottoproblemi  $P(i,j)$   
“Riempire uno zaino di capienza  $j$ , utilizzando un opportuno sottoinsieme dei primi  $i$  oggetti, massimizzando il valore degli oggetti usati”
- Definizione delle soluzioni  $V[i,j]$ 
  - $V[i,j]$  è il massimo valore ottenibile da un sottoinsieme degli oggetti  $\{1,2,\dots,i\}$  in uno zaino che ha capacità  $j$
  - $i = 1,2,\dots,n$
  - $j = 0,1,\dots,P$

La struttura dati che ci verrà in aiuto è una matrice bidimensionale. Il riempimento della cella  $V[i,j]$  vuol dire risolvere il problema  $P(i,j)$ .

#### Casi base

Il primo caso base è quando la capienza dello zaino è 0.

Soluzione: non posso prendere nessun oggetto

Il secondo caso base è quando ho a disposizione un solo oggetto.

Soluzione: lo prendo se ci sta nello zaino

$$\begin{aligned} V[1,j] &= v[1] && \text{se } j \geq p[1] && (\text{c'è spazio per l'oggetto numero 1}) \\ V[1,j] &= 0 && \text{se } j < p[1] && (\text{non c'è spazio per l'oggetto numero 1}) \end{aligned}$$

#### Caso generale

Facendo i casi base riempiamo la prima riga e colonna della nostra matrice, conviene lavorare sulle righe, ovvero analizzare  $i$  oggetti rispetto a un peso fissato  $j$ .

Se il peso è giusto devo decidere se prenderlo o non prenderlo.

Se scartiamo l' $i$ -esimo oggetto vuol dire che sono entrato nello scenario della riga di prima  $V(i-1,j)$

Se prendiamo l' $i$ -esimo oggetto  $\Rightarrow$  il valore si calcola  $V[i] + P(i-1,j-p[i])$   
(il risultato del problema che sta nella riga prima e nella cella più a sinistra del peso.)

Tra i due casi scelgo quello che mi da la soluzione migliore, quindi quello che mi da valore massimo:

$$\max\{V[i-1,j], V[i-1,j-p[i]] + v[i]\}$$

Quindi riassumendo il caso generale con una formula:

$$V[i, j] = \begin{cases} V[i - 1, j] & \text{se } j < p[i] \\ \max\{V[i - 1, j], V[i - 1, j - p[i]] + v[i]\} & \text{se } j \geq p[i] \end{cases}$$

Ecco una definizione matematica di come inizializzare le celle:

$$P(i, j) = \begin{cases} 0 & \text{se } j = 0 \text{ || } (i = 1 \text{ && } P[1] > j) \\ v[1] & \text{se } i = 1 \text{ && } P[1] \leq j \\ P(i - 1, j) & \text{se } P[i] < j \\ \max\{P(i - 1, j), v[i] + P[i - 1, j - P[i]]\} & \text{se } P[i] \geq j \end{cases}$$

### 9.3.2.1 Esempio

$$\begin{array}{l} p = [ \quad 2, \quad 7, \quad 6, \quad 4 \quad ] \\ v = [ \quad 12.7, \quad 6.4, \quad 1.7, \quad 0.3 \quad ] \end{array}$$

|   | 0   | 1   | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   |
|---|-----|-----|------|------|------|------|------|------|------|------|------|
| 1 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 |
| 2 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 19.1 | 19.1 |
| 3 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 14.4 | 19.1 | 19.1 |
| 4 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 13.0 | 13.0 | 14.4 | 19.1 | 19.1 |

|   | 0   | 1   | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   |
|---|-----|-----|------|------|------|------|------|------|------|------|------|
| 1 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 |
| 2 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 19.1 | 19.1 |
| 3 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 12.7 | 14.4 | 19.1 | 19.1 |
| 4 | 0.0 | 0.0 | 12.7 | 12.7 | 12.7 | 12.7 | 13.0 | 13.0 | 14.4 | 19.1 | 19.1 |

$$\begin{aligned} V[3,8] &= \max\{V[2,8], V[2,8 - 6] + 1.7\} \\ &= \max\{12.7, 14.4\} \end{aligned}$$

### 9.3.2.2 Soluzione e trovare gli oggetti presi

La soluzione la si può leggere in basso a destra della matrice, quindi in posizione  $V[n, P]$

Problema: come posso capire quali oggetti sono stati presi?

Si usa una matrice bidimensionale booleana  $K[i, j]$  ausiliaria con le stesse dimensioni di  $V[i, j]$ .  
 $K[i, j] = \text{true} \Leftrightarrow$  l'oggetto  $i$ -esimo fa parte della soluzione del problema  $P(i, j)$  che ha valore  $V[i, j]$ ,

```
integer j ← P;
integer i ← n;
while ( i > 0 ) do
    if ( K[i,j] = true ) then
        stampa "Selezione oggetto ", i
        j ← j - p[i];
    endif;
    i ← i - 1;
endwhile
```

$p = [ \quad 2, \quad 7, \quad 6, \quad 4 \quad ]$   
 $v = [ \quad 12.7, \quad 6.4, \quad 1.7, \quad 0.3 \quad ]$

### 9.3.2.3 Il costo computazionale

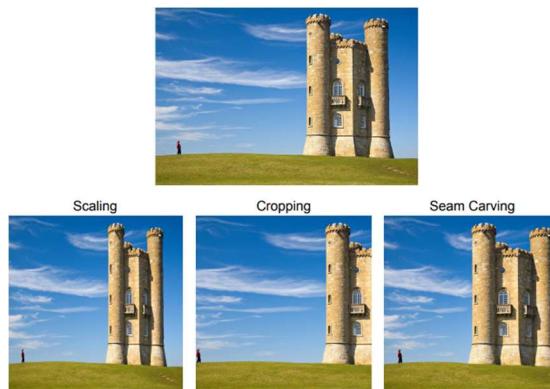
Il costo sarà relativo al riempimento della matrice e allo spazio che occupa.

Il costo, quindi, è la grandezza della matrice

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| 1 | F | F | T | T | T | T | T | T | T | T | T  |
| 2 | F | F | F | F | F | F | F | F | T | T | T  |
| 3 | F | F | F | F | F | F | F | F | T | F | F  |
| 4 | F | F | F | F | F | F | T | T | F | F | F  |

### 9.3.3 Seam Carving

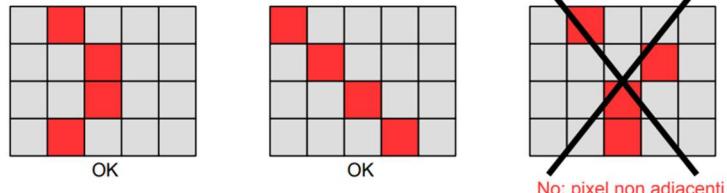
Il Seam Carving è un algoritmo per ridimensionare le immagini senza perdere informazioni importanti, per esempio:



L'algoritmo si basa sul togliere strisce di immagine che hanno poche informazioni e “cucirle” in modo da avere un risultato simile all'immagine originale.

Una cucitura (seam) è un cammino composto da pixel adiacenti di “minima importanza”

**Come determinare le cuciture di peso minimo?**



No: pixel non adiacenti

Assegniamo un peso  $E[i,j] \in [0,1]$  a ogni pixel  $(i,j)$  che indica quanto il pixel sia “importante” (un criterio potrebbe essere quanto un pixel è “diverso” da quelli adiacenti)

L'algoritmo che determina le cuciture di peso minimo verrà strutturato con la PD.

Determinate le cuciture verticali di peso minimo, l'algoritmo procederà così:

- Rimuovere i pixel della cucitura, ottenendo un'immagine  $M \times (N - 1)$
- Ripetere il procedimento fino ad ottenere la larghezza desiderata.

#### › Definizione dei sottoproblemi $P(i,j)$

Determinare una cucitura di peso minimo che termina nel pixel di coordinate  $(i,j)$ .

#### › Definizione delle soluzioni $W[i,j]$

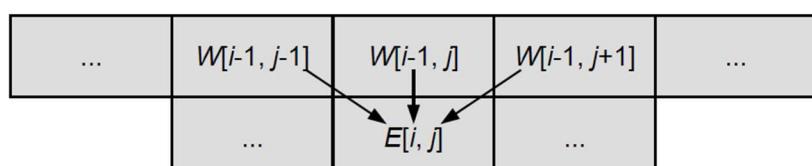
Minimo preso tra tutte le possibili cuciture che terminano nel pixel di coordinate  $(i,j)$ .

#### › Calcolo della soluzione del problema originario

La cucitura di peso minimo avrà peso pari al minimo tra  $\{W[m, 1], \dots, W[M, N]\}$

Casi base  $(i = 1)$

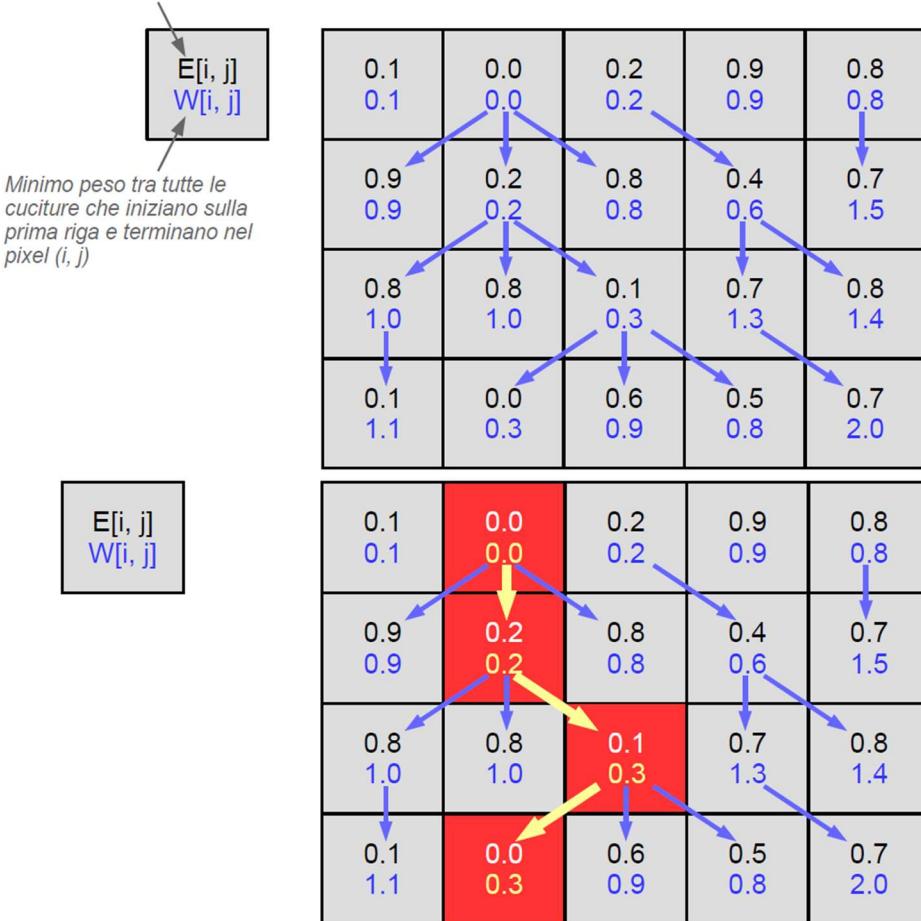
- $W[1,j] = E[1,j] \quad \forall j = 1, \dots, M$



Caso generale  $(i > 1)$

- Se  $j = 1$   
 $W[i,j] = E[i,j] + \min\{W[i-1,j], W[i-1,j+1]\}$
- Se  $1 < j < N$   
 $W[i,j] = E[i,j] + \min\{W[i-1,j-1], W[i-1,j], W[i-1,j+1]\}$
- Se  $j = N$   
 $W[i,j] = E[i,j] + \min\{W[i-1,j-1], W[i-1,j]\}$

**Esempio:** Energia del pixel  $(i, j)$



### 9.3.4 Distanza di Levenshtein

È un algoritmo usato, ad esempio, da uno spell checker che controlla ogni parola che non appartiene al dizionario con quelle ad essa simili per trovare quella più simile.

Per calcolare la similitudine di due stringhe si può usare la distanza di Levenshtein.

Scriveremo dunque un algoritmo `int lev(char s1[n], char s2[n])` che ritorna un numero in base alla similitudine di due stringhe. (basso se sono simili, alto altrimenti).

La distanza di Levenshtein è basata sul concetto di *edit distance* che consiste nel numero di operazioni di editing per trasformare la parola sbagliata in quella giusta. Gli editing ammessi sono:

1. Lasciare immutato il carattere corrente (costo 0)
2. Cancellare un carattere (costo 1)
3. Inserire un carattere (costo 1)
4. Sostituire il carattere corrente con uno diverso (costo 1)

NOTA: il cursore può muoversi solo in una direzione (nel nostro caso solo in avanti), partendo dal primo carattere e applicando un editing per poi spostarsi al successivo.

#### › Definizione dei sottoproblemi $P(i, j)$

Determinare il minor numero di operazioni di editing necessarie per trasformare il prefisso  $S[1..i]$  di  $S$  nel prefisso  $T[1..j]$  di  $T$ .

#### › Definizione delle soluzioni $L[i, j]$

Minor numero di operazioni di editing necessarie per trasformare il prefisso  $S[1..i]$  di  $S$  in  $T[1..j]$  di  $T$ .

#### › Calcolo della soluzione del problema originario

La distanza di Levenshtein tra  $S[1..n]$  e  $T[1..m]$  è il valore  $L[n, m]$

Calcoliamo  $L[n, m]$

Se  $i = 0 \text{ || } j = 0$  (caso base)

Il costo per trasformare una stringa vuota in una non vuota è dato dalla lunghezza della stringa non vuota, quindi  $L[i, j] = \max\{i, j\}$

Se  $i > 0 \text{ && } j > 0$  (caso generico)

- Trasformare  $S[1..i - 1]$  in  $T[1..j]$ , e cancellare l'ultimo carattere  $S[i]$  di  $S$
- Trasformare  $S[1..i]$  in  $T[1..j - 1]$  e inserire l'ultimo carattere  $T[j]$  di  $T$
- Trasformare  $S[1..i - 1]$  in  $T[1..j - 1]$  e cambiare  $S[i]$  in  $T[j]$  se diversi;

Quindi

Se  $S[i] = T[j]$

Analizzeremo dei prefissi delle nostre sottostringhe come sottoproblema base e vedremo che sarà facile identificarli una volta calcolati i primi tempi in tempo costante.

Indichiamo le sottostringhe delle stringhe  $S[1..n]$  e  $T[1..m]$  come  $S[1..i]$  con  $i \in \{0, \dots, n\}$  e  $T[1..j]$  con  $j \in \{0, \dots, m\}$ . Usando  $i = 0 \vee j = 0$  indichiamo le stringhe vuote.

Come per tutti gli algoritmi in programmazione dinamica useremo una matrice  $L[i \times j]$  per salvare i risultati delle chiamate precedenti di  $P$ . Il risultato del problema  $P(n, m)$  sarà  $L[n, m]$

La prima colonna della matrice sarà riempita con i risultati di  $P(i, 0) = i, \forall i = 0, \dots, n$  poiché passare da una stringa lunga  $i$  a una lunga 0 richiede  $i$  operazioni di delete. Analogamente per la prima riga iterando sulla seconda variabile  $j$ .

$$P(i, j) = \begin{cases} \max\{i, j\} & \text{se } i = 0 \vee j = 0 \quad (0) \\ P(i - 1, j - 1) & \text{se } S[i] = T[j] \quad (1) \\ 1 + \min \{P(i - 1, j - 1), P(i - 1, j), P(i, j - 1)\} & \text{se } S[i] \neq T[j] \quad (4, 2, 3) \end{cases}, \quad \forall i = 0, \dots, n, \forall j = 0, \dots, m$$

Esempio:

|     | '''' | L | I | B | R | O |
|-----|------|---|---|---|---|---|
| ''' | 0    | 1 | 2 | 3 | 4 | 5 |
| A   | 1    | 1 | 2 | 3 | 4 | 5 |
| L   | 2    | 1 | 2 | 3 | 4 | 5 |
| B   | 3    | 2 | 2 | 2 | 3 | 4 |
| E   | 4    | 3 | 3 | 3 | 3 | 4 |
| R   | 5    | 4 | 4 | 4 | 3 | 4 |
| O   | 6    | 5 | 5 | 5 | 4 | 3 |

Minimo numero di operazioni necessarie per trasformare ALBE in LIB

Distanza di Levenshtein tra ALBERO e LIBRO

### 9.3.5 Conclusioni:

La programmazione dinamica è una tecnica algoritmica estremamente potente,

che però va applicata (dove applicabile) con **disciplina**

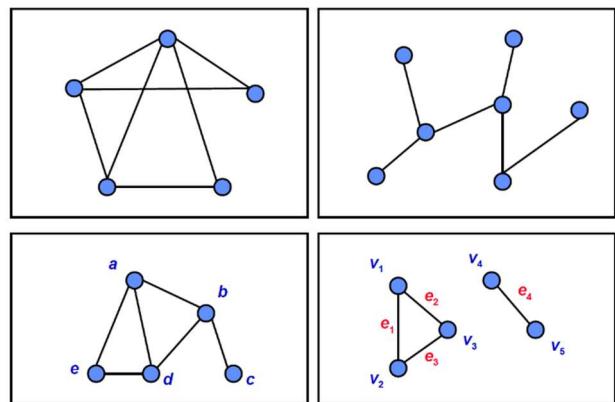
- Identificare i sottoproblemi
- Definire le soluzioni dei sottoproblemi
- Calcolare le soluzioni nei casi semplici
- Calcolare le soluzioni nel caso generale

## 10 GRAFI

Nell'immagine d'esempio di seguito:

1. Il primo grafo è *ciclico* (cioè, posso raggiungere un nodo in modo circolare)
2. Il secondo può essere orientato come un albero (e viene detto *albero libero*)

I nodi/vertici possono essere collegati in modo arbitrario e possono avere un peso (come anche gli archi, cioè i collegamenti).



Gli archi possono essere o meno orientati, cioè, hanno un verso oppure sono bidirezionali

### 10.1.1 Problemi sui grafi

1. Visite
  - a. In ampiezza (BFS), usata per identificare il cammino di lunghezza minima da una singola sorgente
  - b. Profondità (DFS)
2. Alberi di copertura minima
3. Cammini minimi da una singola sorgente o tra tutte le coppie di vertici

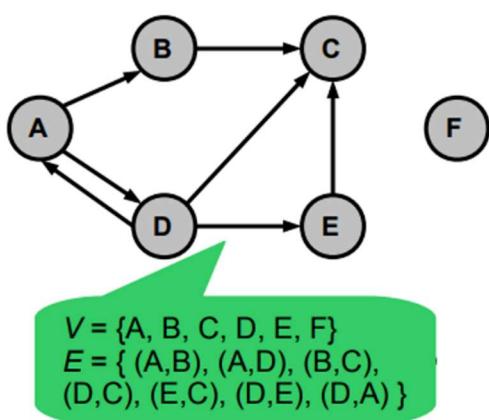
## 10.2 DEFINIZIONI

### 10.2.1 Grafi orientati e non orientati

#### 10.2.1.1 Orientati

Un grafo  $G$  è una coppia  $(V, E)$ , dove:

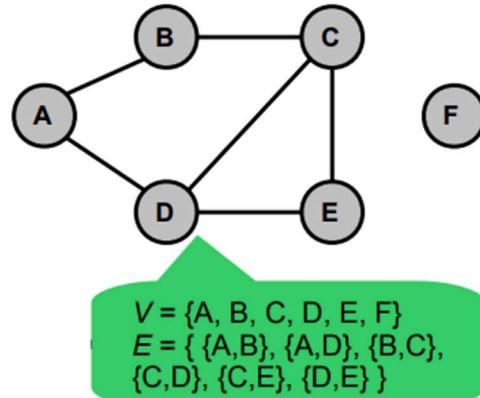
- $V$  è l'insieme finito dei vertici
- $E$  è l'insieme degli archi **orientati** che collega gli elementi contenuti in  $V$ .  
Perciò se si vuole un collegamento a doppio senso tra due nodi  $A$  e  $D$ , bisogna inserire le coppie  $(A, D)$  e  $(D, A)$  dentro a  $E$



#### 10.2.1.2 Non orientati

Un grafo non orientato  $G$  è una coppia  $(V, E)$  dove:

- $V$  è l'insieme finito dei vertici
- $E$  è l'insieme finito degli archi, dove usiamo la notazione degli insiemi  $\{A, D\}$  perché con questa notazione ignoriamo l'ordine nel quale gli elementi appaiono.

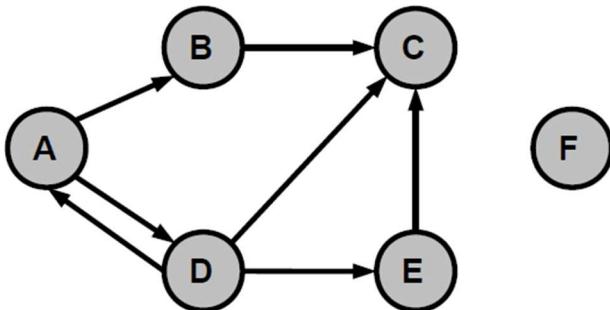


### 10.2.2 Adiacenza e incidenza

In un grafo *orientato* l'arco  $(v, w)$  si dice **incidente** da  $v$  in  $w$  (mettiamo in relazione un arco coi vertici).

Un vertice  $w$  è **adiacente** a  $v \Leftrightarrow \exists(v, w) \in E$  (relazioniamo due vertici controllando se esiste un arco)

In un grafo non orientato la relazione di adiacenza tra vertici è simmetrica.



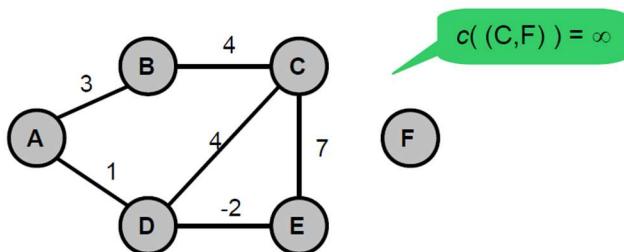
- $(A, B)$  è incidente da  $A$  in  $B$
- $(A, D)$  è incidente da  $A$  a  $D$
- $(D, A)$  è incidente da  $D$  a  $A$
- $B$  è adiacente ad  $A$
- $C$  è adiacente a  $B, D, E$
- $A$  è adiacente a  $D$  e viceversa
- $B$  non è adiacente a  $D$  e a  $C$
- $F$  non è adiacente ad alcun vertice

### 10.2.3 Grafi pesati

In alcuni casi ogni arco ha un **peso** (o costo) associato.

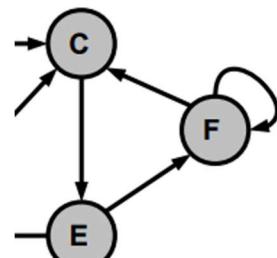
Il costo può essere determinato tramite una funzione di costo  $c: E \rightarrow \mathbb{R}$

Quando tra due vertici non esiste un arco si dice che il costo è infinito.



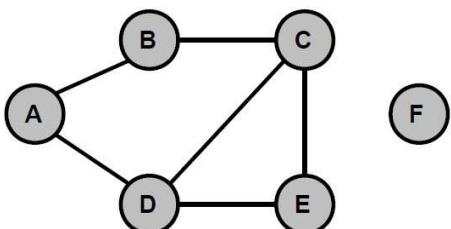
### 10.2.4 Cappi

Ovvero archi che puntano allo stesso vertice.



### 10.2.5 Grado

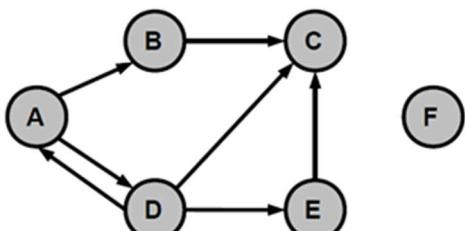
In un grafo *non orientato*, il grado di un vertice è il numero di archi che partono da esso.



- $A, B$  ed  $E$  hanno grado 2
- $C$  e  $D$  hanno grado 3
- $F$  ha grado 0

Nel caso di grafi *orientati*, si possono distinguere due declinazioni:

- Grado entrante: numero archi incidenti in esso
- Grado totale: la somma di grado entrante e uscente
- Grado uscente: numero archi incidenti da esso



| Nodo:                      | grado uscente | grado entrante      |
|----------------------------|---------------|---------------------|
| A                          | 2             | 1                   |
| B                          | 1             | 1                   |
| C                          | 0             | 3                   |
| D                          | 3             | 1                   |
| <i>A e C hanno grado 3</i> |               | <i>C ha grado 3</i> |

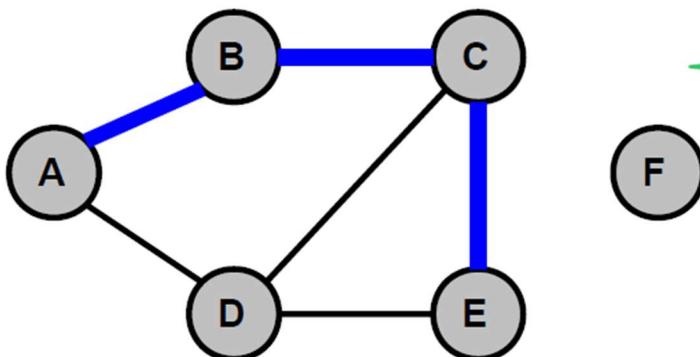
*B ha grado 2**D ha grado 4*

### 10.2.6 Cammini

Un **cammino** in  $G = (V, E)$  è una sequenza di vertici  $\langle w_0, \dots, w_k \rangle$  tale che  $\{w_i, w_{i+1}\} \in E, \forall i = 0, \dots, k - 1$  (quindi è una sequenza di archi che può essere percorsa).

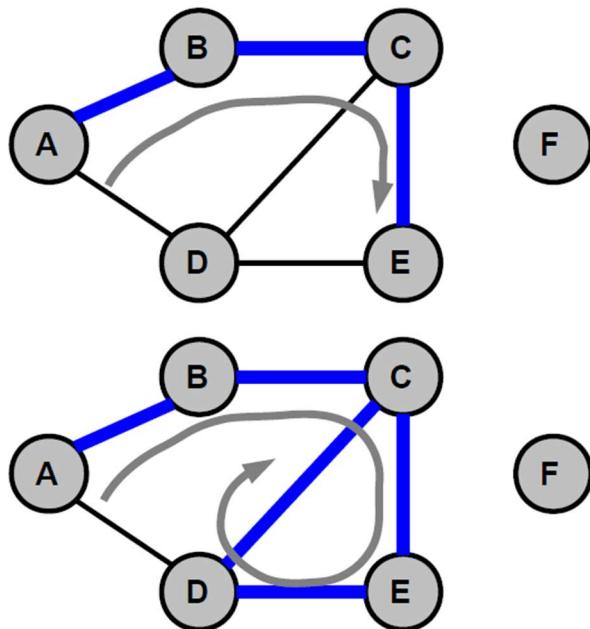
Il cammino  $\langle w_0, \dots, w_k \rangle$  contiene  $w_0, \dots, w_k$  e gli archi  $\{w_0, w_1\}, \dots, \{w_{k-1}, w_k\}$ .

La **lunghezza del cammino** è il numero di archi attraversati.



**<A, B, C, E>** è un cammino di lunghezza 3

Un cammino si dice **semplice** se tutti i suoi vertici sono distinti (compaiono una sola volta nella sequenza)



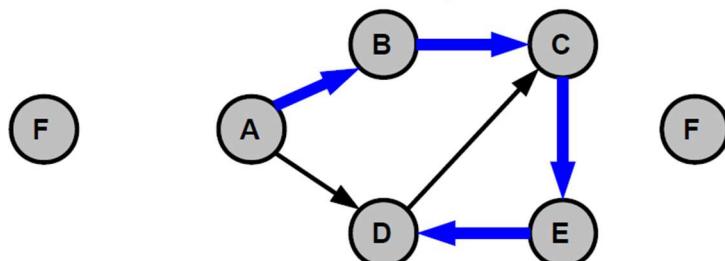
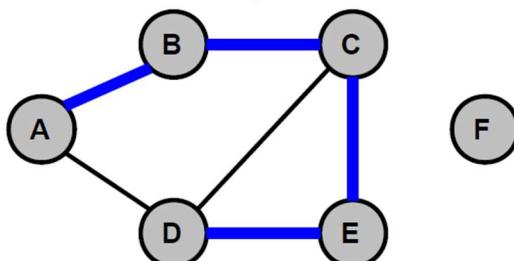
Il cammino **<A, B, C, E>** è semplice...

... ma il cammino **<A, B, C, E, D, C>** non è semplice, poiché C è ripetuto

Se esiste un cammino  $c$  tra i vertici  $v$  e  $w$ , si dice che w è raggiungibile da v tramite c.

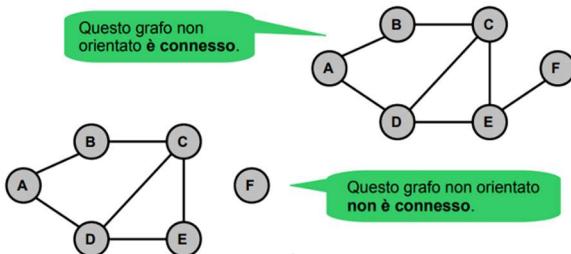
A è raggiungibile da D e viceversa

D è raggiungibile da A ma non viceversa



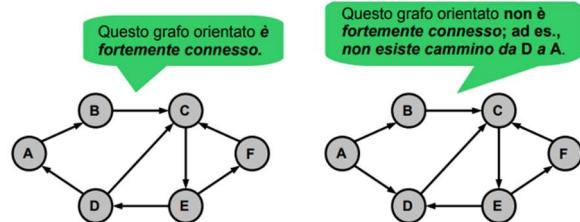
### 10.2.7 Grafi connessi e fortemente connessi

Se  $G$  è un grafo *non orientato*, diciamo che  $G$  è **connesso** se esiste almeno un cammino tra tutti i vertici



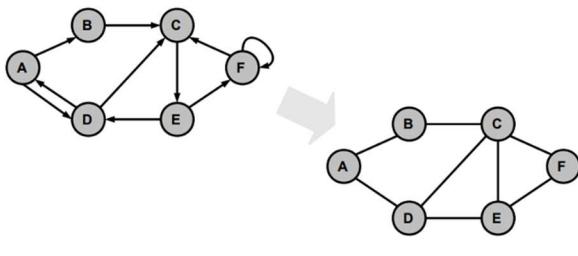
Se  $G$  è un grafo *orientato*, diciamo che  $G$  è **fortemente connesso** se esiste un cammino da ogni vertice a ogni altro vertice.

(posso raggiungere ogni vertice da qualunque vertice io parta)

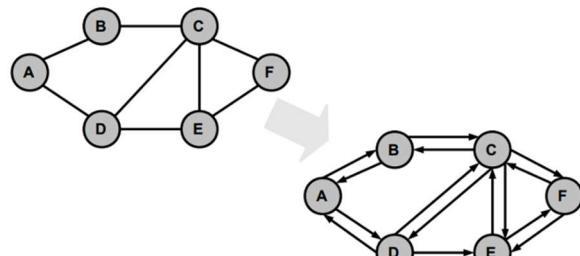


### 10.2.8 Versione orientata e non orientata

Se  $G$  è un grafo *orientato*, allora il grafo ottenuto ignorando la direzione degli archi e i cappi è detto **versione non orientata di  $G$** .

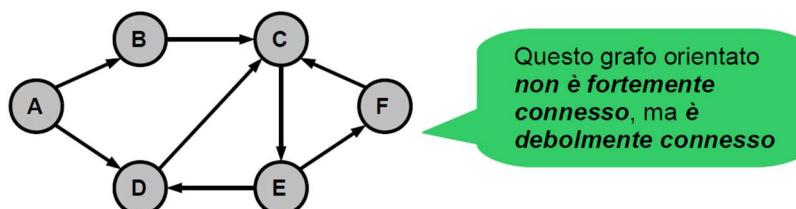


Se  $G$  è un grafo *non orientato*, allora il grafo ottenuto sostituendo ogni arco con due archi orientati opposti è detto **versione orientata di  $G$** .



### 10.2.9 Grafi debolmente connessi

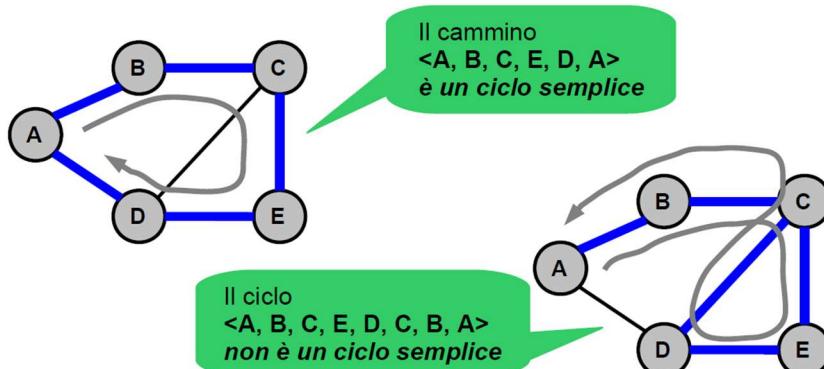
Se  $G$  è un grafo orientato che *NON* è fortemente connesso, ma la sua versione non orientata è connessa, diciamo che  $G$  è **debolmente connesso**



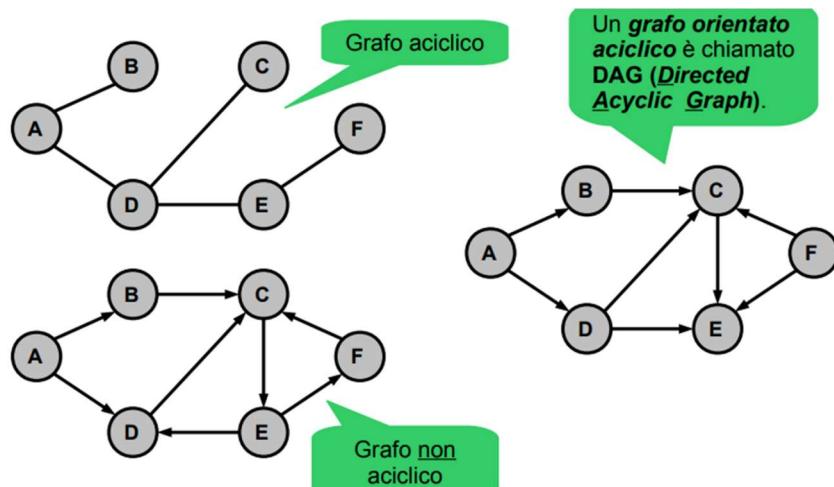
### 10.2.10 Grafi ciclici e aciclici

Un **ciclo** in un grafo *orientato* è un cammino di lunghezza  $\geq 1$  tale che  $w_0 = w_n$

Un **ciclo semplice**, è un ciclo in cui i nodi  $w_1, \dots, w_{n-1}$  sono tutti distinti.

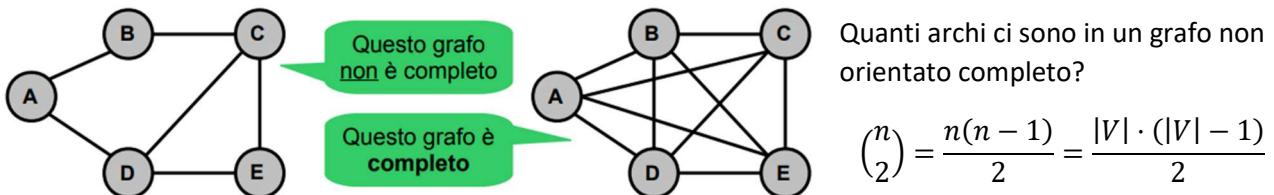


Un grafo con **cicli** è detto **ciclico**, invece uno senza cicli è detto **aciclico**



### 10.2.11 Grafo completo

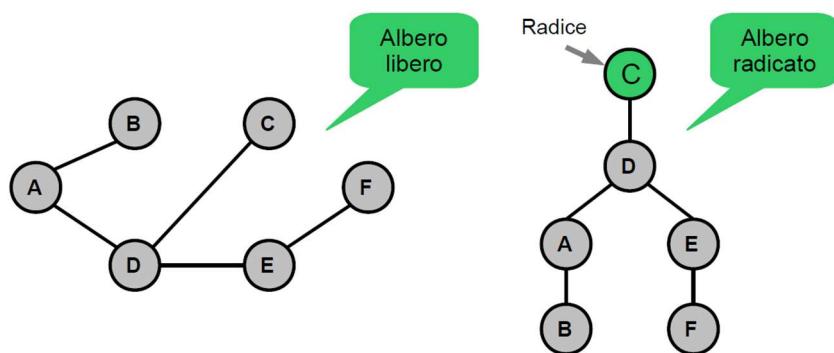
Un grafo **non orientato completo** è un grafo che ha un arco tra ogni coppia di vertici.



### 10.2.12 Albero libero

Un **albero libero** è un grafo **non orientato connesso, aciclico**.

Se un vertice è detto radice  $\Rightarrow$  otteniamo un albero radicato



## 10.3 FUNZIONI,

ovvero operazioni che la struttura dati deve supportare

```
NumVertici() -> int
NumArchi() -> int
Grado(vert v) -> int
archiIncidenti(vert v) -> (arc, arc, ..., arc)
estremi(arc e) -> (vert, vert)
opposto(vert x, arc e) -> vert
```

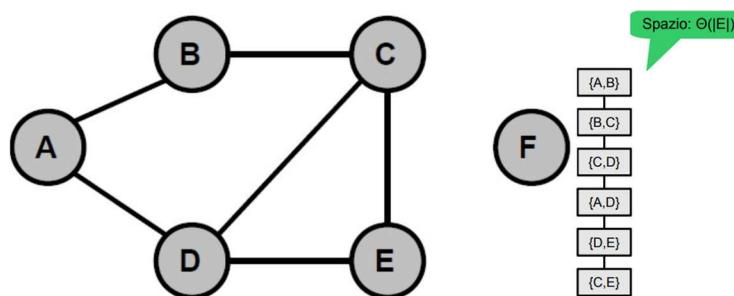
```
sonoAdiacenti(vert x, vert y) ->
bool
aggiungiVertice(vert v)
aggiungiArco(vert x, vert y)
rimuoviVertice(vert v)
rimuoviArco(arc e)
```

## 10.4 IMPLEMENTAZIONI

$m$  viene usato per indicare la cardinalità di  $E$ , invece  $n$  per la cardinalità di  $V$ .

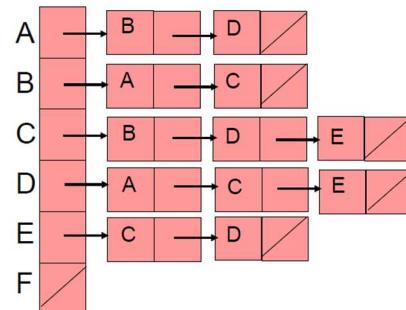
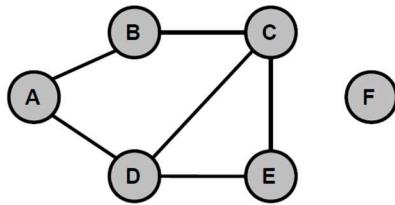
### 10.4.1 Liste di archi (grafo non orientato)

|   |        |
|---|--------|
| grado(vert v) -> int                      | $O(m)$ |
| archiIncidenti(vert v) -> (arc, ..., arc) | $O(m)$ |
| sonoAdiacenti(vert x, vert y) -> bool     | $O(m)$ |
| aggiungiVertice(vert v)                   | $O(1)$ |
| aggiungiArco(arc e)                       | $O(1)$ |
| rimuoviVertice(vert v)                    | $O(m)$ |
| rimuoviArco(arc e)                        | $O(1)$ |

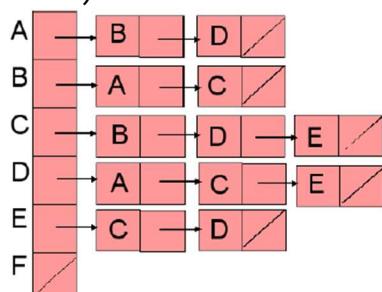


### 10.4.2 Liste di incidenza

$$v.\text{adj} = \{ w \mid \{v,w\} \in E \}$$



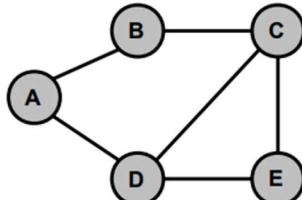
|   |                                   |
|---|-----------------------------------|
| grado(vert v) -> int                      | $O(\delta(v))$                    |
| archiIncidenti(vert v) -> (arc, ..., arc) | $O(\delta(v))$                    |
| sonoAdiacenti(vert x, vert y) -> bool     | $O(\min\{\delta(x), \delta(y)\})$ |
| aggiungiVertice(vert v)                   | $O(1)$                            |
| aggiungiArco(arc e)                       | $O(1)$                            |
| rimuoviVertice(vert v)                    | $O(m)$                            |
| rimuoviArco(arc e)                        | $O(\delta(x)+\delta(y))$          |



Indichiamo con  
 $n=|V|$ ,  $m=|E|$ ,  
 $\delta(x)$  = grado del nodo x  
(spiegato in seguito)

### 10.4.3 Matrice di adiacenza (grafo non orientato)

$$M(u, v) = \begin{cases} 1 & \text{se } \{u, v\} \in E \\ 0 & \text{altrimenti} \end{cases}$$

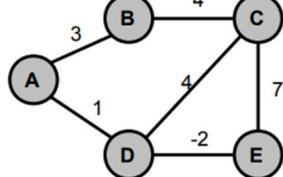
Spazio:  $\Theta(|V|^2)$ 

$$M = \begin{pmatrix} & \textcolor{blue}{A} & \textcolor{blue}{B} & \textcolor{blue}{C} & \textcolor{blue}{D} & \textcolor{blue}{E} & \textcolor{blue}{F} \\ \textcolor{blue}{A} & 0 & 1 & 0 & 1 & 0 & 0 \\ \textcolor{blue}{B} & 1 & 0 & 1 & 0 & 0 & 0 \\ \textcolor{blue}{C} & 0 & 1 & 0 & 1 & 1 & 0 \\ \textcolor{blue}{D} & 1 & 0 & 1 & 0 & 1 & 0 \\ \textcolor{blue}{E} & 0 & 0 & 1 & 1 & 0 & 0 \\ \textcolor{blue}{F} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

|   |            |
|---|------------|
| grado(vert v) -> int                      | 0(n)       |
| archiIncidenti(vert v) -> (arc, ..., arc) | 0(n)       |
| sonoAdiacenti(vert x, vert y) -> bool     | 0(1)       |
| aggiungiVertice(vert v)                   | 0( $n^2$ ) |
| aggiungiArco(arc e)                       | 0(1)       |
| rimuoviVertice(vert v)                    | 0( $n^2$ ) |
| rimuoviArco(arc e)                        | 0(1)       |

In grafi non orientati pesati

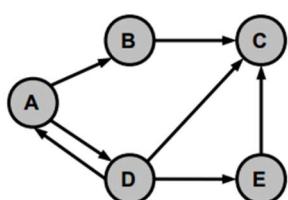
$$M(u, v) = \begin{cases} c(u, v) & \text{se } \{u, v\} \in E \\ \infty & \text{altrimenti} \end{cases}$$

Spazio:  $\Theta(|V|^2)$ 

$$M = \begin{pmatrix} & \textcolor{blue}{A} & \textcolor{blue}{B} & \textcolor{blue}{C} & \textcolor{blue}{D} & \textcolor{blue}{E} & \textcolor{blue}{F} \\ \textcolor{blue}{A} & \infty & 3 & \infty & 1 & \infty & \infty \\ \textcolor{blue}{B} & 3 & \infty & 4 & \infty & \infty & \infty \\ \textcolor{blue}{C} & \infty & 4 & \infty & 4 & 7 & \infty \\ \textcolor{blue}{D} & 1 & \infty & 4 & \infty & -2 & \infty \\ \textcolor{blue}{E} & \infty & \infty & 7 & -2 & \infty & \infty \\ \textcolor{blue}{F} & \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

### 10.4.4 Matrice di adiacenza (grafo orientato)

$$M(u, v) = \begin{cases} 1 & \text{se } (u, v) \in E \\ 0 & \text{altrimenti} \end{cases}$$

Spazio:  $\Theta(|V|^2)$ 

$$M = \begin{pmatrix} & \textcolor{blue}{A} & \textcolor{blue}{B} & \textcolor{blue}{C} & \textcolor{blue}{D} & \textcolor{blue}{E} & \textcolor{blue}{F} \\ \textcolor{blue}{A} & 0 & 1 & 0 & 1 & 0 & 0 \\ \textcolor{blue}{B} & 0 & 0 & 1 & 0 & 0 & 0 \\ \textcolor{blue}{C} & 0 & 0 & 0 & 0 & 0 & 0 \\ \textcolor{blue}{D} & 1 & 0 & 1 & 0 & 1 & 0 \\ \textcolor{blue}{E} & 0 & 0 & 1 & 0 & 0 & 0 \\ \textcolor{blue}{F} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

## 10.5 VISITE GRAFI

### DEF. PROBLEMA:

Dato un grafo  $G = \langle V, E \rangle$  e un vertice  $s \in V$  (detto **sorgente**), visitate ogni vertice raggiungibile nel grafo dal vertice  $s$ . Ogni nodo deve essere visitato una volta sola

- # Visita in ampiezza (BFS – Breadth-first-search)
  - Visita i nodi “espandendo” la frontiera fra nodi scoperti / da scoprire
  - Es: *Cammini di lunghezza minima da singola sorgente*
- # Visita in profondità (DFS – depth-first-search)
  - Visita i nodi andando “il più lontano possibile” nel grafo
  - Es: *Ordinamento topologico*

Possiamo utilizzare algoritmi per la visita in ampiezza e profondità sugli alberi, tramite una coda o uno stack ma dobbiamo curarci di segnare i vertici già visitati in quanto i grafi possono essere ciclici e il nostro algoritmo potrebbe visitare alcuni nodi più volte e andare in loop.

Perciò marcheremo ogni nodo con tre diversi tipi di stato:

1. **inesplorato** il nodo non è ancora stato visitato
2. **aperto** il nodo è uno di quelli attualmente visitati
3. **chiuso** il nodo è già stato esplorato

L'algoritmo mantiene un sottoinsieme frontiera  $F \subseteq T$

- Se un nodo  $v$  sta in  $T - F$ , significa che tutti gli archi incidenti sono stati esplorati  $(v \text{ è chiuso})$
- Se un nodo  $v$  sta in  $F$ , non tutti gli archi sono stati esplorati  $(v \text{ è aperto})$
- Se un nodo non sta in  $T$  allora è inesplorato

### 10.5.1 Visita in ampiezza (BFS)

```

1. algoritmo BFS(grafo G, vertice s) -> albero
2.   for each v in V {
3.     v.mark := false;
4.     T = s;
5.     F = new Queue();
6.     F.enqueue(s);
7.     s.mark = true;
8.     s.dist = 0;
9.     while (f != null) {
10.       u = F.dequeue(); // visita il vertice u
11.       for each v adiacente a u {
12.         if (not v.mark) {
13.           v.mark = true;
14.           T = T U v;
15.           F.enqueue(v);
16.           v.parent = u;
17.           v.dist = u.dist + 1;
18.         }
19.       }
20.     }
21.   }
22.   return T;

```

- insieme  $F$  gestito tramite una coda
- $v.\text{mark}$  è la marcatura del nodo  $v$
- $v.\text{dist}$  è la distanza del nodo  $v$  dal vertice  $s$
- $v.\text{parent}$  è il padre di  $v$  nell'albero  $T$
- ritorna l'albero  $T$  costruito dalla visita

### Costo computazionale

Notiamo che:

- > Il ciclo while viene eseguito una volta per ogni vertice visitato
- > Una esecuzione del for “legge” gli archi incidenti dal vertice che si sta visitando
- > Ciascun arco viene letto:
  - Una volta nei grafi orientati
  - Al più due volte in quelli non orientati

### Proprietà della BFS

- Visita i nodi a distanze crescenti dalla sorgente
- Genera un albero BF (breadth-first)
- Calcola la distanza minima da  $s$  a tutti i vertici raggiungibili

Complessità:

- #  **$O(n + m)$**  per le liste di adiacenza
- #  **$O(n^2)$**  per le matrici di adiacenza

## Applicazioni

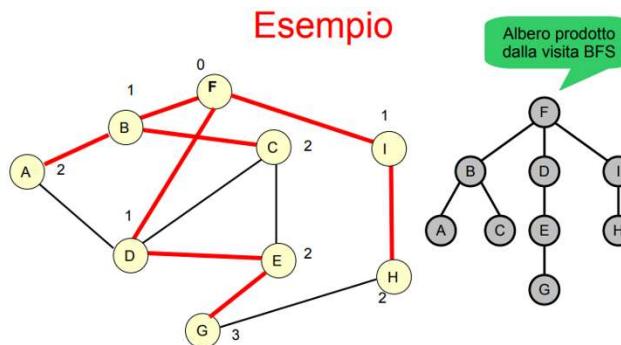
La visita BFS può essere utilizzata per ottenere il percorso più breve (minor numero di archi traversati) fra due vertici

Per esempio, il seguente pseudocodice stampa un cammino più breve tra due nodi  $s$  e  $v$

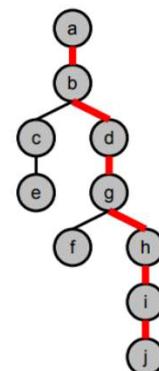
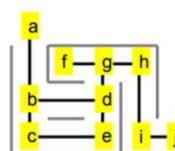
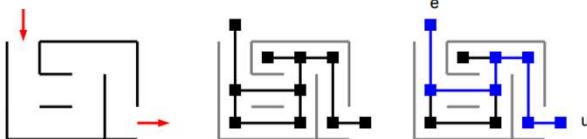
- Il grafo  $G$  è stato precedentemente visitato con l'algoritmo BFS a partire da  $s$  e con output l'albero della visita  $T$ .

```

1. algoritmo print-path(G, s, v) {
2.   if (v == s) {
3.     print s;
4.   } else if (v.parent = null) {
5.     print "no path from s to v";
6.   } else {
7.     print-path(G, s, v.parent);
8.     print v;
9.   }
10. }
```



Percorso più breve per uscire dal labirinto?



### 10.5.2 Visita in profondità (DFS)

La visita in profondità viene utilizzata per coprire l'intero grafo e non solo i nodi raggiungibili da una singola sorgente, al contrario della visita in ampiezza che visita solo i nodi raggiungibili.

L'output sarà una foresta  $G_\pi = (V, E_\pi)$ , contenente un insieme di alberi DF. Inoltre vengono aggiunte informazioni sul tempo di visita (tempo di scoperta di un nodo e tempo di "terminazione" di un nodo)

- Versione ricorsiva
- $Time$  è una variabile globale che contiene il numero di "passi" dell'algoritmo
- $v.dt$  (*discovery time*): tempo in cui il nodo è stato scoperto
- $v.ft$  (*finish time*): tempo in cui la visita del nodo termina

```

1. global time = 0; // variabile globale
2.
3. void DFS (Grafo G) {
4.   for each u in V {
5.     u.mark = white;
6.     u.parent = NULL;
7.   } for each u in V {
8.     if (u.mark == white) {
9.       DFS-visit(u);
10.    }
11. }
12.
13. void DFS-visit(Node u) {
14.   u.mark = gray;
15.   time++;
16.   u.dt = time;
17.   for each v adiacente a u {
18.     if (v.mark == white) {
19.       v.parent = u;
20.       DFS-visit(v);
21.     }
22.   }
23.   visita(u);
24.   time++;
25.   u.ft = time;
26.   u.mark = black;
27. }
```

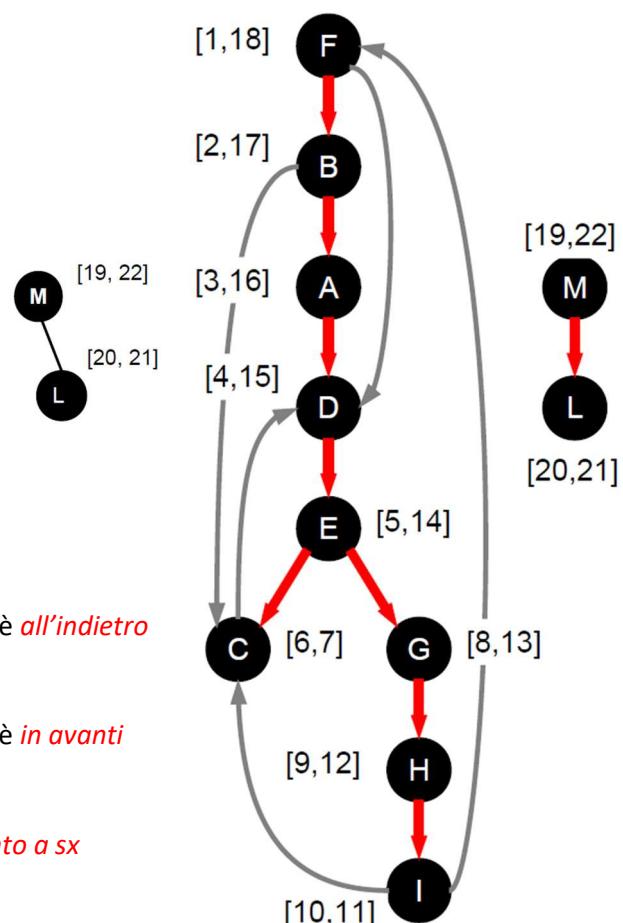
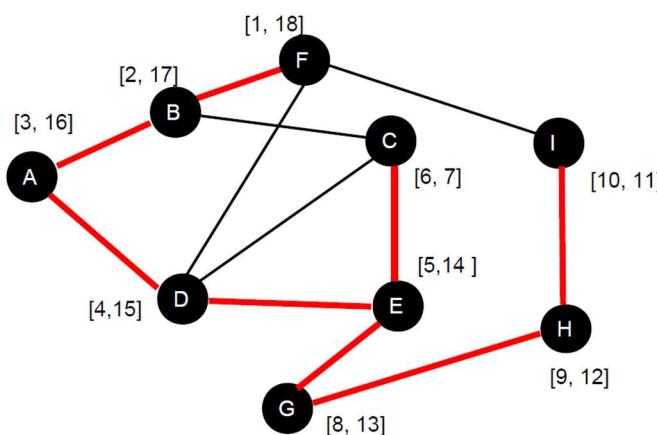
### Proprietà (teoria delle parentesi)

In una qualsiasi visita di profondità di un grafo  $G = (V, E)$ , per ogni coppia di vertici  $u, v$ , una sola delle seguenti condizioni è vera:

- › Gli intervalli  $[u.dt, u.ft]$  e  $[v.dt, v.ft]$  sono disgiunti  
 $\Rightarrow u, v$  non sono discendenti l'uno dell'altro nella foresta DF
- › L'intervallo  $[u.dt, u.ft]$  è interamente contenuto in  $[v.dt, v.ft]$   
 $\Rightarrow u$  è discendente di  $v$  in un albero DF
- › L'intervallo  $[v.dt, v.ft]$  è interamente contenuto in  $[u.dt, u.ft]$   
 $\Rightarrow v$  è discendente in  $u$  in un albero DF

#### Corollario:

- Il vertice  $v$  è un discendente del vertice  $u$  nella foresta DF  $\Leftrightarrow u.dt < v.dt < v.ft < u.ft$



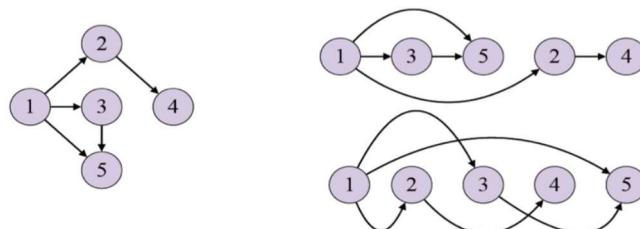
Per quanto riguarda i grafi orientati invece:

- › Se  $v.dt < u.dt$  e  $u.ft < v.ft \Rightarrow$  l'arco  $\{u, v\}$  è *all'indietro*  
 eg.  $\{v = F \quad u = I\} \Rightarrow \begin{cases} 1_v < 10_u \\ 11_u < 18_v \end{cases}$
- › Se  $u.dt < v.dt$  e  $v.ft < u.ft \Rightarrow$  l'arco  $\{u, v\}$  è *in avanti*  
 eg.  $\{v = F \quad u = D\} \Rightarrow \begin{cases} 1_u < 4_v \\ 15_v < 18_u \end{cases}$
- › Se  $v.ft < u.dt \Rightarrow$  l'arco  $\{u, v\}$  è *di attraversamento a sx*
- › NOTA: non possono esistere altri casi!

#### 10.5.2.1 Ordinamento topologico (in DAG)

Dato un DAG  $G$ , un ordinamento topologico su  $G$  è un ordinamento lineare dei suoi vertici tale per cui:

- Se  $G$  contiene l'arco  $(u, v) \Rightarrow u$  compare prima di  $v$  nell'ordinamento
- Per transitività, ne consegue che se  $v$  è raggiungibile da  $u \Rightarrow u$  compare prima nell'ordinamento



#### Algoritmo per ordinamento topologico:

- Si effettua un DFS
- L'operazione di visita aggiunge il nodo alla testa di una lista "at finish time"
- Restituisce la lista di vertici

Come output avremo una sequenza ordinata di vertice in ordine inverso di finish time.

### 10.5.3 Componenti connesse

Due vertici  $u$  e  $v$  sono connessi se e solo se  $u$  è raggiungibile da  $v$ .

La relazione “ $u$  è raggiungibile da  $v$ ” è di equivalenza:

- › Riflessiva:
  - $u$  è raggiungibile da sé stesso
- › Simmetrica
  - Se  $u$  è raggiungibile da  $v$ , allora  $v$  è raggiungibile da  $u$
- › Transitiva
  - Se  $u$  è raggiungibile da  $v$  e  $v$  è raggiungibile da  $w \Rightarrow u$  è raggiungibile da  $w$

La relazione di raggiungibilità induce quindi un partizionamento dei vertici in componenti connesse.

### 10.5.4 Componenti fortemente connesse

Un grafo *orientato*  $G$  è fortemente connesso se ogni coppia di vertici è connessa da un cammino.

Due vertici  $u$  e  $v$  sono fortemente connessi se e solo se esiste un cammino orientato che connette  $u$  con  $v$  (e viceversa).

La relazione di connettività forte è di equivalenza:

- › Riflessiva:
  - $u$  è raggiungibile da sé stesso
- › Simmetrica:
  - Se  $u$  è fortemente connesso a  $v$ , allora  $v$  è fortemente connesso a  $u$ .
- › Transitiva:
  - Se  $u$  è fortemente connesso a  $v$  e  $v$  è fortemente connesso a  $w \Rightarrow u$  è fortemente connesso a  $w$

La relazione di connessione forte induce un partizionamento dei vertici in componenti fortemente connesse.

Visto che  $y$  appartiene alla stessa componente fortemente connessa di  $x$  se e soltanto se esiste un cammino da  $x$  a  $y$  ed esiste un cammino da  $y$  a  $x$ . Allora la componente connessa a cui  $x$  appartiene è costituita dai vertici che sono raggiungibili da  $x$  e da cui  $x$  è raggiungibile

Basterà quindi calcolare:

- $D(x) :=$  insieme dei discendenti del nodo  $x$  (tutti i nodi raggiungibili da  $x$ )
- $A(x) :=$  insieme degli antenati del nodo  $x$  (tutti i nodi dai quali si raggiunge  $x$ )

e avremo che la componente fortemente connessa a cui appartiene  $x$  è l'intersezione  $D(x) \cap A(x)$

?

Come calcolare  $D(x)$  ?

$D(x)$  include i nodi raggiungibili da una visita usando  $x$  come sorgente

?

Come calcolare  $A(x)$  ?

È sufficiente invertire la direzione di tutti gli archi ed effettuare una nuova visita usando ancora  $x$  come sorgente.

Nota: il tempo di calcolo di  $A(x)$  o  $D(x)$  richiede tempo  $O(n + m)$

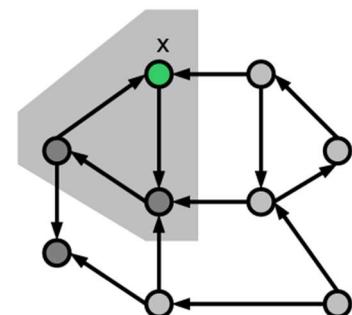
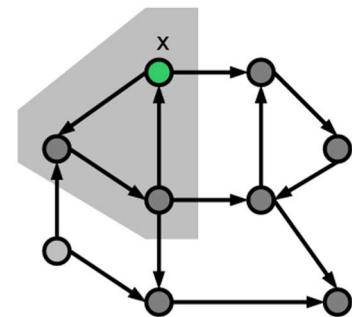
```

1. algoritmo SCC(Grafo G, nodo x) {
2.   L := lista vuota di nodi
3.   (1) esegui BFS(G, x) marcando i nodi visitati
4.   (2) calcola il grafo trasposto  $G^T$  (inverte direzione archi)
5.   (3) esegui BFS( $G^T$ , x) mettendo in L i nodi
6.     visitati che sono stati marcati durante (1)
7.   Return L;
8. }
```

(1), (2) e (3) costano tutti  $O(n + m)$

Per calcolare tutte le SCC di un grafo  $G$  è necessario eseguire l'algoritmo SCC( $G, x$ ) per ogni nodo  $x \in V$

(ogni esecuzione di SCC( $G, x$ ) costa  $O(n + m)$ )



Costo complessivo  $O(nm + n^2)$

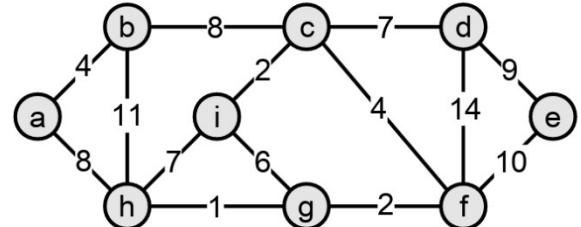
## 10.6 MINIMUM SPANNING TREE

È un problema di notevole importanza e consiste nel determinare come interconnettere diversi elementi tra loro minimizzando certi vincoli sulle connessioni.

Un esempio (classico) è la progettazione di circuiti elettronici dove si vuole minimizzare la quantità di filo elettrico per collegare fra loro i diversi componenti.

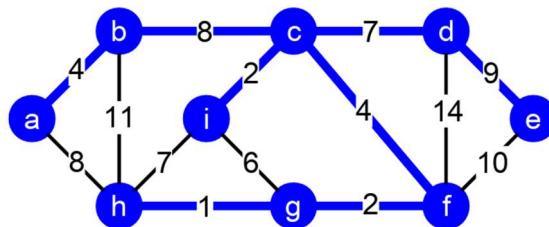
Input:

- $G = \langle V, E \rangle$ , un grafo non orientato e connesso
- $w: V \times V \rightarrow R$  una funzione peso
  - Se  $\{u, v\} \in E \Rightarrow w(u, v)$  è il peso dell'arco  $\{u, v\}$
  - Se  $\{u, v\} \notin E \Rightarrow w(u, v) = \infty$
- Poiché  $G$  non è orientato  $w(u, v) = w(v, u)$



Dato un grafo  $G$ , non orientato e connesso, un **albero di copertura** di  $G$  è un sottografo  $T = \langle V, E \rangle$  tale che:

- $T$  è un albero
- $E_T \subseteq E$
- $T$  contiene tutti i nodi di  $G$



Il problema, quindi, è:

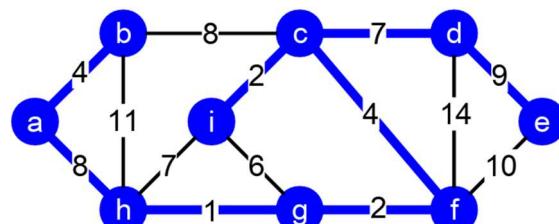
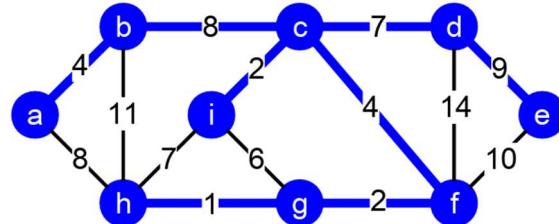
Dato un grafo orientato e pesato, definire un albero di copertura, minimo, ovvero un albero di copertura t.c.

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

Sia minimo tra tutti i possibili alberi di copertura.

OSS:

Il minimum spanning tree non è necessariamente unico:



### Metodo generico per calcolare MST:

L'idea è di accrescere un sottoinsieme di  $T$  archi in modo tale che venga rispettata la seguente condizione:

- $T$  è un sottoinsieme di qualche albero di copertura minimo
- Un arco  $\{u, v\}$  è detto **sicuro** per  $T$  se  $T \cup \{u, v\}$  è ancora un sottoinsieme per qualche MST

```

1. Tree generic-MST(Grafo G=<V, E, w>) {
2.   Tree T ← albero generico
3.   while T non forma un albero di copertura do
4.     trova un arco sicuro {u, v}
5.     T ← T ∪ {u, v}
6.   endwhile
7.   return T
8. }
```

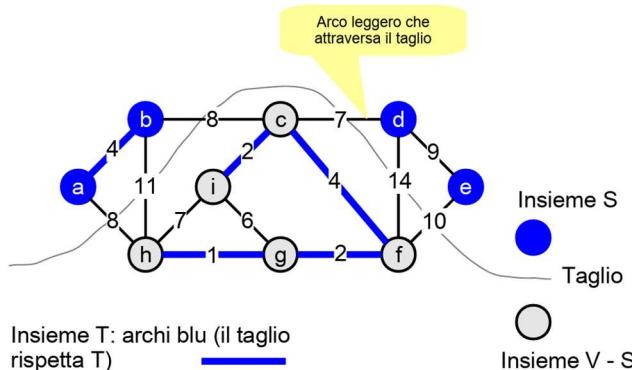
Per caratterizzare gli archi sicuri dobbiamo introdurre alcune definizioni:

- Un **taglio**  $(S, V - S)$  di un grafo non orientato  $G = (V, E)$  è una partizione di  $V$  in due sottoinsiemi disgiunti
- Un arco  $\{u, v\}$  **attraversa il taglio** se  $u \in S$  e  $v \in V - S$
- Un taglio **rispetta** un insieme di archi  $T$  se nessun arco di  $T$  attraversa il taglio
- Un arco che attraversa il taglio è **leggero** se il suo peso è minimo fra i pesi degli archi che attraversano un taglio

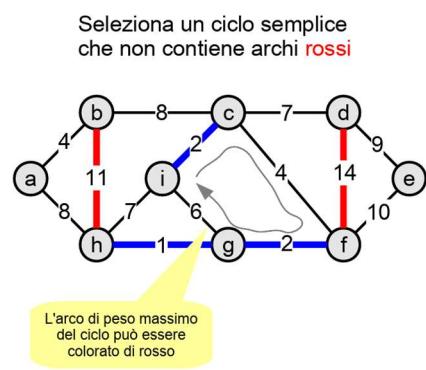
Regole del ciclo e del taglio:

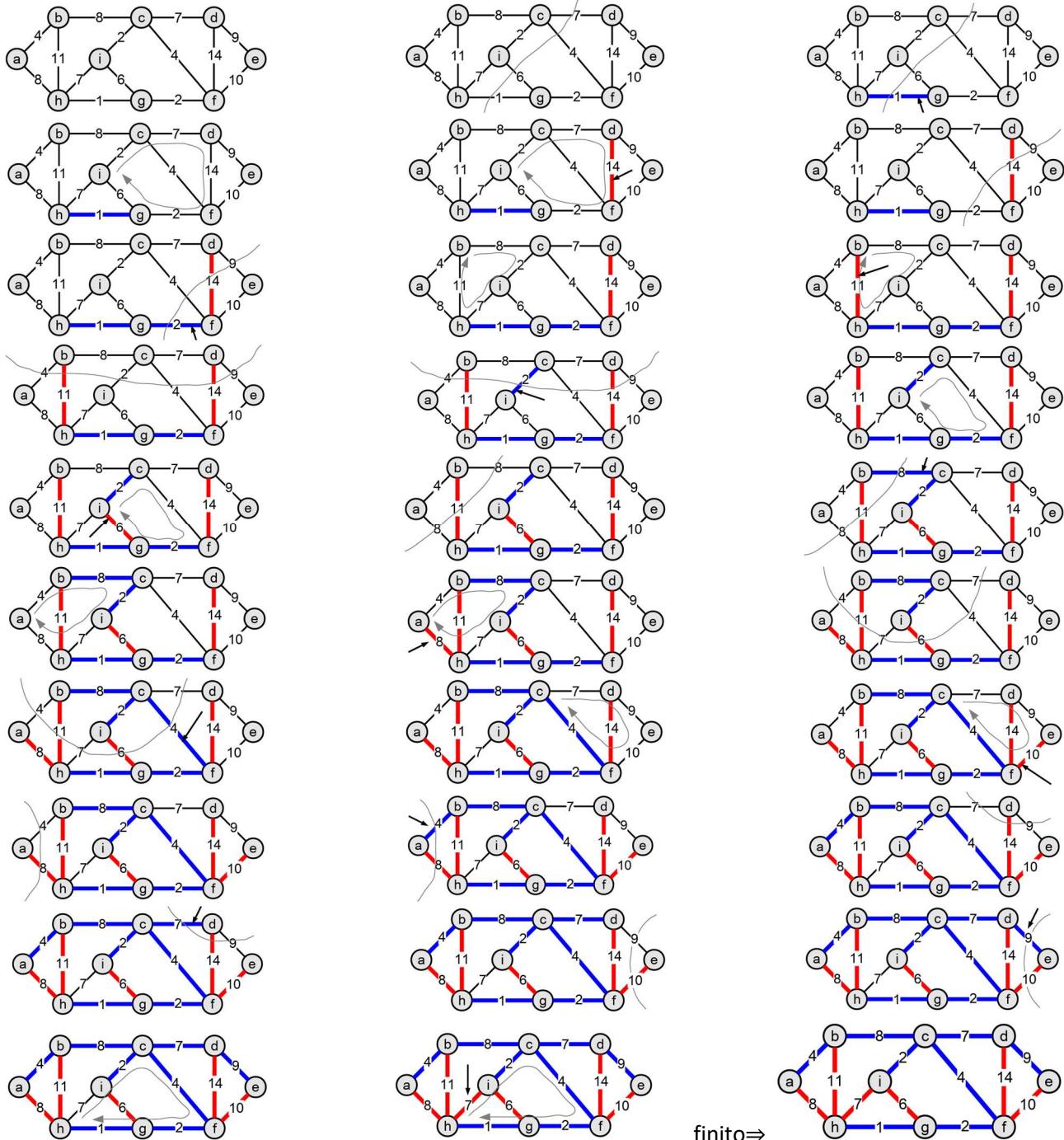
- **Regola del taglio:**  
Scegli un taglio in  $G$  che rispetta gli archi blu. Tra tutti gli archi non colorati che attraversano il taglio seleziona uno leggero (di peso minimo) e coloralo di blu.
- **Regola del ciclo:**  
Scegli un ciclo semplice in  $G$  che non contenga archi rossi. Tra tutti gli archi non colorati del ciclo, seleziona un arco di costo massimo e coloralo di rosso.
- Si può costruire un MST usando queste regole:  
Costruisce un MST applicando in successione una delle due regole precedenti.

### Applicazione regola del taglio



### Applicazione regola del ciclo





### Algoritmo di Kruskal:

Questo non è un algoritmo, poiché non viene indicato deterministicamente come applicare le regole.

Kruskal fissa un ordine di applicazione delle regole:

- Idea: ingrandire sottoinsiemi disgiunti di un albero di copertura minima connettendoli fra loro fino ad avere l'albero finale
- Si considerano gli archi in ordine decrescente di peso
  - Se l'arco  $e = \{u, v\}$  connette due alberi blu  $\Rightarrow$  lo si colora di blu. Altrimenti lo si colora di rosso
- L'algoritmo è greedy perché a ogni passo si aggiunge alla foresta un arco con il peso minimo.

L'ordinamento richiede:  $O(m \log m) = O(m \log n^2) = O(m \log n)$  dove  $m$  num. archi,  $n$  num. nodi  
il tempo di esecuzione dipende dalla realizzazione per insiemi disgiunti. Se usiamo QuickUnion la sequenza costa in tutto  $O(n + m \log n + m)$

$$\text{Totale: } O(2m \log n + 2n) = O(m \log n)$$

**Algoritmo di Prim:**

Utilizza solo la regola del taglio; l'ordine di applicazione della regola dipende da un nodo  $r$  detto *radice*, da cui si assume di far partire l'algoritmo.

Si procede mantenendo in un singolo albero  $T$  che viene fatto via via "crescere".

- L'albero parte da un nodo arbitrario  $r$  e cresce fino a quando non ricopre tutti i vertici
- Ad ogni passo viene aggiunto l'arco di peso minimo che collega un nodo già raggiunto dell'albero con uno non ancora raggiunto.

Costo computazionale:

Utilizzando una coda di priorità basata su min-heap:

- $n \text{ deleteMin}()$  costano:  $O(n \log n)$
- $n \text{ insert}()$  costano:  $O(n \log n)$
- $O(m) \text{ decreaseKey}()$  costano:  $O(m \log n)$

$$\text{Totale} = O(n \log n + n \log n + m \log n) = O(m \log n)$$

## 10.7 CAMMINI MINIMI

Consideriamo un grafo orientato  $G = \langle V, E \rangle$  in cui ogni arco  $(x, y) \in E$  sia associato un costo  $w(x, y)$ . Il costo di un cammino  $\pi = (v_0, v_1, \dots, v_k)$  che collega il nodo  $v_0$  con  $v_k$  è definito come:

$$w(\pi) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

Data una coppia di nodi  $v_0$  e  $v_k$ , vogliamo trovare (se esiste) un cammino  $\pi_{v_0, v_k}^*$  di costo minimo tra tutti i cammini che vanno da  $v_0$  a  $v_k$ .

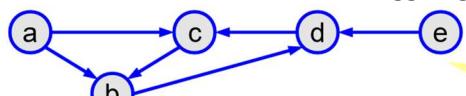
Abbiamo **diversi modi di formulare il problema**:

1. Cammino di costo minimo fra una singola coppia di nodi  $u$  e  $v$ 
  - Determinare, se esiste, un cammino di costo minimo  $\pi_{uv}^*$  da  $u$  verso  $v$
2. Single-source shortest path
  - Determinare cammini di costo minimo da un nodo sorgente  $s$  a tutti i nodi raggiungibili da  $s$
3. All-pairs shortest paths
  - Determinare cammini di costo minimo tra ogni coppia di nodi  $u, v$

### Osservazioni:

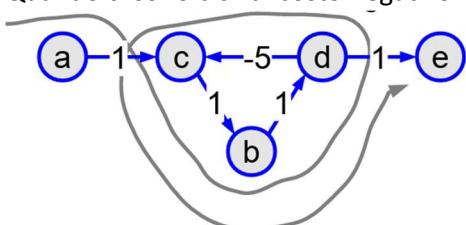
In quali situazioni *non* esiste un cammino di costo minimo?

- Quando la destinazione non è raggiungibile



Non esiste alcun cammino che connette  $a$  con  $e$

- Quando ci sono cicli di costo negativo



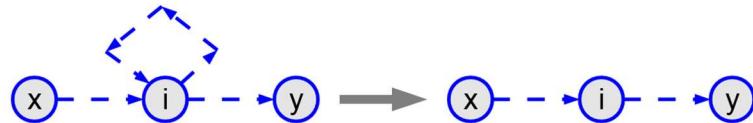
È sempre possibile trovare un cammino di costo inferiore che connette  $a$  con  $e$

**Esistenza:**

Sia  $G = \langle V, E \rangle$  un grafo orientato con funzione di peso  $w$ . Se non ci sono cicli negativi  $\Rightarrow$  fra ogni coppia di vertici connessi in  $G$  esiste sempre un cammino semplice di costo minimo.

*Proof:*

Possiamo sempre trasformare un cammino in un cammino semplice (privo di cicli)



Ogni volta che si rimuove un ciclo, il costo diminuisce (o rimane uguale)

Il numero di cammini semplici è finito, esiste un minimo.

**Prop. Sottostruttura ottima:**

Sia  $G = \langle V, E \rangle$  un grafo orientato con funzione di peso  $w$ . Allora *ogni sotto-cammino di un cammino di costo minimo in  $G$  è a sua volta un cammino di costo minimo*.

*Proof:*

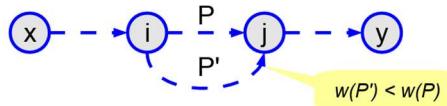
Consideriamo un cammino minimo  $\pi_{xy}^*$  da  $x$  a  $y$

Siano  $i$  e  $j$  due nodi intermedi

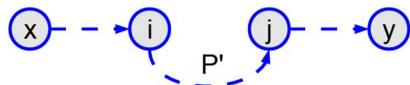
Dimostriamo che il sotto-cammino di  $\pi_{xy}^*$  che collega  $i$  e  $j$  è un cammino minimo tra  $i$  e  $j$ .



Supponiamo per assurdo che esista un cammino  $P'$  tra  $i$  e  $j$  di costo strettamente inferiore a  $P$



Ma allora potremmo costruire un cammino tra  $x$  e  $y$  di costo inferiore a  $\pi_{xy}^*$ , il che è assurdo perché avevamo fatto l'ipotesi che  $\pi_{xy}^*$  fosse il cammino di costo minimo.

**Distanza tra vertici in un grafo:**

Sia  $G = \langle V, E \rangle$  un grafo orientato con funzione di peso  $w$ . La distanza  $d_{xy}$  tra  $x$  e  $y$  in  $G$  è il *costo di un cammino di costo minimo che li connette*;  $+\infty$  se questo cammino non esiste.

$$d_{xy} = \begin{cases} w(\pi_{xy}^*) & \text{se esiste un cammino di costo minimo } \pi_{xy}^* \\ +\infty & \text{altrimenti} \end{cases}$$

- Nota:  $d_{vv} = 0 \quad \forall v \in V$
- Nota: vale la diseguaglianza triangolare

$$d_{xz} \leq d_{xy} + d_{yz}$$

**Tecnica del rilassamento:**

Supponiamo di mantenere una stima  $D_{sv} \geq d_{sv}$  della lunghezza del cammino di costo minimo tra  $s$  e  $v$ . Effettuiamo dei passi di "rilassamento", riducendo progressivamente la stima finché si ha  $D_{sv} = d_{sv}$

```
if (Dsu + w(u,v) < Dsv) then Dsv ← Dsu + w(u,v)
```

### 10.7.1 Algoritmo di Bellman e Ford

Consideriamo un cammino di costo minimo inizialmente ignoto

$$\pi_{sv_k}^* = (s, v_1, \dots, v_k)$$

Sappiamo che  $d_{sv_k} = d_{sv_{k-1}} + w(v_{k-1}, v_k)$

Da cui partendo da  $D_{ss} = 0$  (e  $D_{st} = \infty$ , per  $t \neq s$ ) potremmo effettuare i seguenti passi di rilassamento:

$$\begin{aligned} D_{sv_1} &\leftarrow D_{ss} + w(s, v_1) \\ D_{sv_2} &\leftarrow D_{sv_1} + w(v_1, v_2) \\ &\vdots \\ D_{sv_k} &\leftarrow D_{sv_{k-1}} + w(v_{k-1}, v_k) \end{aligned}$$

Problema: noi non conosciamo gli archi del cammino minimo  $\pi_{sv_k}^*$  ne il loro ordine, quindi non possiamo fare il rilassamento nell'ordine corretto.

Però se eseguiamo *per ogni arco*  $(u, v)$

```
if (Dsu + w(u, v) < Dsv) then Dsv ← Dsu + w(u, v)
```

sicuramente includeremo anche il primo passo di rilassamento “corretto”

$$D_{sv_1} \leftarrow D_{ss} + w(s, v_1)$$

Ad ogni passo consideriamo tutti gli  $m$  archi del grafo  $(u, v)$  ed effettuiamo il passo di rilassamento

Dopo  $n - 1$  iterazioni (tante quanti i possibili vertici di destinazione dei cammini che partono da  $s$ ) siamo sicuri di aver calcolato tutti i valori  $D_{sv_k}$  corretti.

```

1. double [1..n] BellmanFord(Grafo G=(V,E,w), int s)
2. int n ← G.numNodi();
3. int pred[1..n], v, u;
4. for v ← 1 to n do
5.   D[v] ← ∞
6.   pred[v] ← 1
7. endfor
8. D[s] ← 0
9. for int i ← 1 to n-1 do
10.   for each (u,v) in E do
11.     if ( D[u] + w(u,v) < D[v] ) then
12.       D[v] ← D[u] + w(u,v);
13.       pred[v] ← u;
14.     endif
15.   endfor
16. endfor
17. // eventuale controllo per cicli negativi
18. Return D;
```

I nodi del grafo sono identificati dagli interi 1...n

$D[v]$  = (stima della) distanza del nodo  $v$  dalla sorgente

$pred[v]$  = predecessore del nodo  $v$  sul cammino di costo minimo che collega  $s$  con  $v$

L'algoritmo di Bellman e Ford determina i cammini di costo minimo anche in presenza di archi con peso negativo.

Però non devono esistere cicli di peso negativo

Il controllo seguente da fare al termine dell'algoritmo di Bellman e Ford, determina se esistono cicli negativi

```

1. // eventuale controllo per cicli negativi
2. for each (u,v) in E do
3.   if ( D[u] + w(u,v) < D[v] ) then
4.     error "il grafo contiene cicli negativi"
5.   endif
6. endfor
```

Nel caso in cui tutti i pesi siano non negativi, esiste un algoritmo più efficiente

### 10.7.2 Algoritmo di Dijkstra

Determina i cammini di costo minimo da singola sorgente *nel caso in cui tutti gli archi abbiano costi*  $\geq 0$

#### Lemma (Dijkstra)

Sia  $G = \langle V, E \rangle$  un grafo orientato con funzione di costo  $w$

- i costi degli archi devono essere  $\geq 0$

Sia  $T$  una parte dell'albero dei cammini di costo minimo radicato in  $s$

-  $T$  rappresenta porzioni di cammini di costo minimo che partono da  $s$

Allora l'arco  $(u, v)$  con  $u \in V(T)$  e  $v \notin V(T)$  che minimizza la quantità  $d_{su} + w(u, v)$  appartiene a un cammino minimo da  $s$  a  $v$ .

#### Dimostrazione:

Supponiamo per assurdo che  $(u, v)$  non appartenga a un cammino di costo minimo tra  $s$  e  $v$

$$\Rightarrow d_{su} + w(u, v) > d_{sv}$$

Quindi deve esistere  $\pi_{sv}^*$  che porta da  $s$  in  $v$  senza passare per  $(u, v)$  con costo inferiore a  $d_{su} + w(u, v)$

Il cammino  $\pi_{sv}^*$  si compone in  $\pi_{sy}^*$  e  $\pi_{yv}^*$  e  $\pi_{yv}^*$ , con  $y$  primo nodo fuori  $T$  attraversato dal cammino minimo.

$$\Rightarrow d_{sv} = d_{sx} + w(x, y) + d_{yv}$$

Per ipotesi (lemma di Dijkstra), l'arco  $(u, v)$  è quello che, tra tutti gli archi che collegano un vertice in  $T$  con uno ancora non in  $T$ , minimizza la somma  $d_{su} + w(u, v)$

$$\text{In particolare: } d_{su} + w(u, v) \leq d_{sx} + w(x, y)$$

Riassumendo abbiamo:

- (1)  $d_{su} + w(u, v) > d_{sv}$
- (2)  $d_{sv} = d_{sx} + w(x, y) + d_{yv}$
- (3)  $d_{su} + w(u, v) \leq d_{sx} + w(x, y)$

Combinando (1), (2) e (3) otteniamo:

$$\begin{aligned} d_{su} + w(u, v) &> d_{sx} + w(x, y) + d_{yv} \\ &\geq d_{sx} + w(x, y) \\ &\geq d_{su} + w(u, v) \end{aligned}$$

Assurdo!

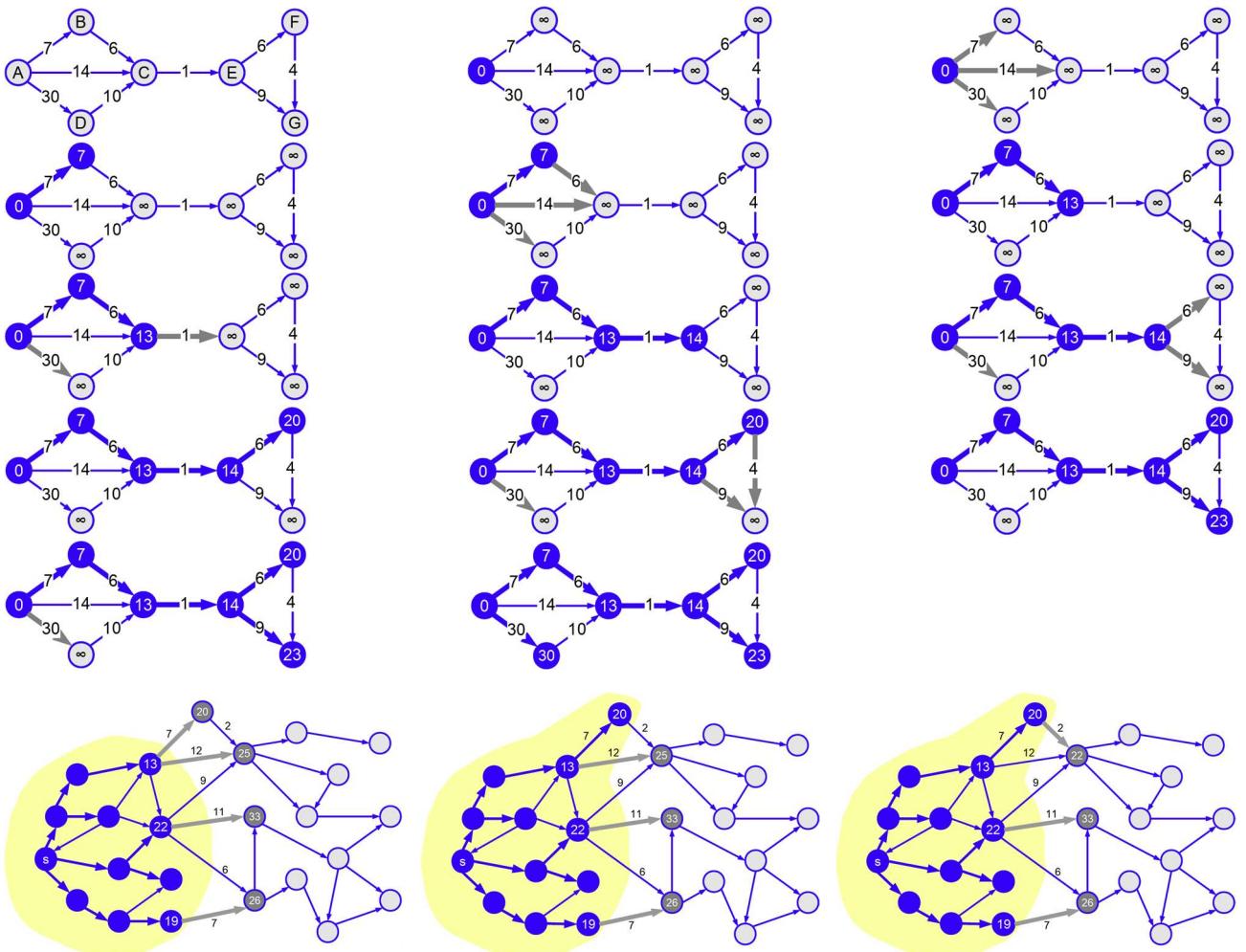
## Algoritmo di Dijkstra generico

```

1. double[1..n] DijkstraGenerico (Grafo G=(V,E,w), int s)
2.   int n ← G.numNodi();
3.   int pred[1..n], u, v;
4.   double D[1..n];
5.   for v ← 1 to n do
6.     D[v] ← +∞;
7.     pred[v] ← -1;
8.   endfor
9.   D[s] ← 0;
10.  while (non ho visitato tutti i nodi raggiungibili da s) do
11.    Trova l'arco (u,v) incidente su T con D[u] + w(u,v) minimo
12.    D[v] ← D[u] + w(u,v);
13.    pred[v] ← u;
14.  endwhile
15.  return D;

```

## Esempio:



```
1. double[1..n] Dijkstra(Grafo G=(V,E,w), int s)
2.   int n ← G.numNodi();
3.   int pred[1..n], v, u;
4.   double D[1..n]
5.   for v ← 1 to n do
6.     D[v] ← +∞
7.     pred[v] ← -1;
8.   endfor
9.   D[s] ← 0;
10.  CodaPriorita<int, double> Q;
11.  Q.insert(s, D[s]);
12.  while (not Q.isEmpty()) do
13.    u ← Q.find();
14.    Q.deleteMin();
15.    for each v adiacente a u do
16.      if (D[v] == +∞) then
17.        D[v] ← D[u] + w(u,v);
18.        Q.insert(v, D[v]);
19.        Pred[v] ← u;
20.      elseif (D[u] + w(u,v) < D[v]) then
21.        Q.decreaseKey(v, D[u] + w(u,v));
22.        D[v] ← D[u] + w(u,v);
23.        Pred[v] ← u;
24.      endif
25.    endfor
26.  endwhile
27.  return D;
```

*Trova e rimuovi il  
nodo con distanza  
minima*

*Somiglia all'algoritmo di Prim (MST),  
ma usa una priorità diversa*

Rendi  $D[u] + w(u,v)$  la  
nuova distanza di  $v$   
da  $s$

## Analisi dell'algoritmo di Dijkstra

- L'inizializzazione ha costo  $O(n)$
- Le operazioni `find()` e `deleteMin()` hanno costo  $O(\log n)$  e sono eseguite al più  $n$  volte
  - Una volta che un nodo è stato estratto dalla coda di priorità non verrà più reinserito
- Le operazioni `insert()` e `decreaseKey()` hanno costo  $O(\log n)$  e sono eseguite al più  $m$  volte
  - Una volta per ogni arco
- Totale:

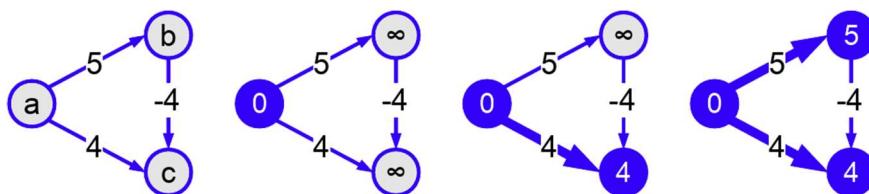
$$O((n+m) \log n) = O(m \log n)$$

Se tutti i nodi sono raggiungibili dalla sorgente

Osservazione

Affinché l'algoritmo di Dijkstra funzioni correttamente è essenziale che *i pesi degli archi siano tutti  $\geq 0$*

- Esempio di funzionamento errato



Il cammino minimo da  $a \rightarrow c$  non è  $(a, c)$  ma  $(a, b, c)$  che ha costo 1

### 10.7.3 Algoritmo di Floyd e Warshall

Si può applicare a grafi orientati con costi arbitrari (anche negativi), purché non ci siano cicli negativi  
(è basato sulla programmazione dinamica)

- Sia  $V = \{1, 2, \dots, n\}$
- Sia  $D_{xy}^k$  la distanza minima dal nodo  $x$  al nodo  $y$ , nell'ipotesi in cui gli eventuali nodi intermedi possano appartenere esclusivamente all'insieme  $\{1, \dots, k\}$
- La soluzione al nostro problema è  $D_{xy}^n$  per ogni coppia di nodi  $x$  e  $y$

## Inizializzazione

$D_{xy}^0$  è la distanza minima tra  $x$  e  $y$  nell'ipotesi di non poter passare per alcun nodo intermedio

Posso calcolare  $D_{xy}^0$  come

$$D_{xy}^0 = \begin{cases} 0 & \text{se } x = y \\ w(x, y) & \text{se } (x, y) \in E \\ \infty & \text{se } (x, y) \notin E \end{cases}$$

## Caso generale

Per andare da  $x$  a  $y$  usando solo nodi intermedi in  $\{1, \dots, k\}$  ho due possibilità

- Non passo mai per il nodo  $k$ . La distanza in tal caso è  $D_{xy}^{k-1}$
- Passo per il nodo  $k$ . Per la proprietà di sottostruttura ottima la distanza in tal caso è  $D_{xk}^{k-1} + D_{ky}^{k-1}$

Quindi

$$D_{xy}^k = \min\{D_{xy}^{k-1}, D_{xk}^{k-1} + D_{ky}^{k-1}\}$$

## Algoritmo di Floyd e Warshall

```

1. double[1..n,1..n] FloydWarshall(G=(V,E,w))
2. int n < G.numNodi();
3. for x < 1 to n do
4.   for y < 1 to n do
5.     if (x == y) then D[x,y,0] <= 0;
6.     elseif ((x,y)∈E) then D[x,y,0] <= w(x,y);
7.     else D[x,y,0] ←+∞;
8.     endif
9.   endfor
10. endfor
11. for k < 1 to n do
12.   for x < 1 to n do
13.     for y < 1 to n do
14.       D[x,y,k] ← D[x,y,k-1];
15.       if (D[x,k,k-1] + D[k,y,k-1] < D[x,y,k]) then D[x,y,k] ← D[x,k,k-1] + D[k,y,k-1];
16.       endif
17.     endfor
18.   endfor
19. endfor
20. for x < 1 to n do      // eventuale controllo per cicli negativi
21.   if (D[x,x] < 0) then
22.     error "Il grafo contiene cicli negativi"
23.   endif
24. endfor
25. return D[1..n, 1..n, n];

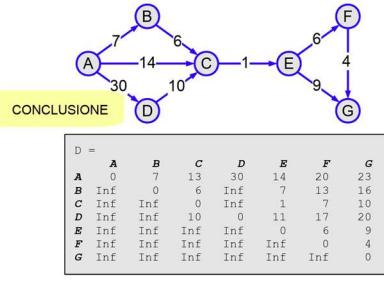
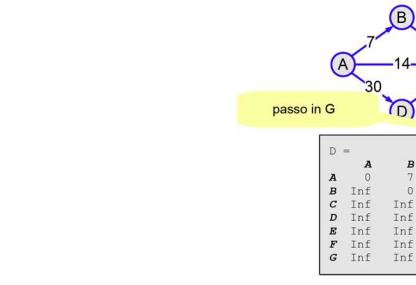
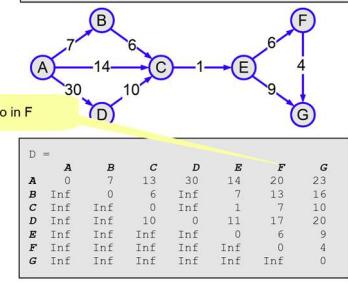
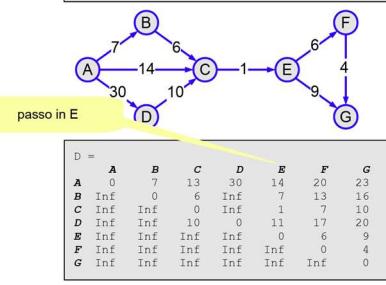
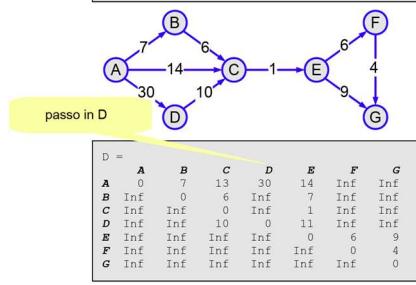
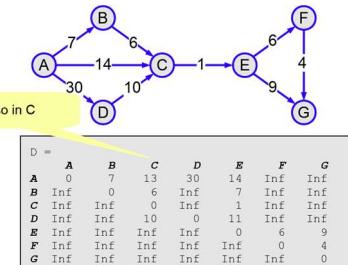
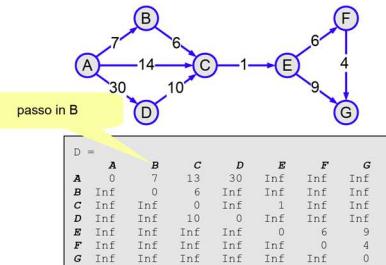
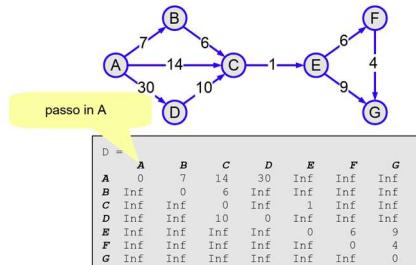
```

$$D_{xy}^k = \min\{D_{xy}^{k-1}, D_{xk}^{k-1} + D_{ky}^{k-1}\}$$

Costo: tempo  $O(n^3)$ , spazio  $O(n^3)$

(c'è una versione ottimizzata per spazio sulle slide)

Esempio:



## 11 TEORIA DELLA NP-COMPLETEZZA

---

Un problema  $Q$  può essere definito come una relazione

$$Q \subseteq I \times S$$

- $I$  è l'insieme delle istanze di ingresso
- $S$  è l'insieme delle soluzioni

Possiamo immaginare  $Q$  come un predicato che, dato in ingresso una istanza di input  $x \in I$  e una soluzione  $s \subseteq S$ , restituisce:

- 1 se  $(x, s) \in Q$  ( $s$  è soluzione del problema  $Q$  sull'istanza  $x$ )
- 0 altrimenti ( $s$  non è soluzione del problema  $Q$  sull'istanza  $x$ )

Abbiamo diverse tipologie di problemi:

- Problemi di decisione
  - Problemi che richiedono una risposta binaria ( $S = \{0,1\}$ )
  - Es: "Un dato grafo  $x$  è connesso?"
  - Es: "Un elemento  $x$  è contenuto in un dizionario?"
- Problemi di ricerca
  - Data una istanza  $x$ , restituire una soluzione  $s$  tale che  $(x, s) \in Q$
  - Es: "Trovare un albero di copertura per il grafo  $x$ "
- Problemi di ottimizzazione
  - Data una istanza  $x$ , restituire la "miglior" soluzione  $s$
  - Es: "Trovare un minimo albero di copertura per il grafo  $x$ "

### 11.1 CLASSI DI COMPLESSITÀ

Data una funzione  $f(n)$ , chiamiamo  $Time(f(n))$  (e rispettivamente  $Space(f(n))$ ) l'insieme di tutti i **problemi decisionali** che possono essere risolti in tempo (risp. in spazio)  $O(f(n))$ , dove  $n$  indica la dimensione dell'input.

Ossia tutti i problemi che ammettono un algoritmo  $A$  t.c.:

- per ogni istanza di input  $x$ ,  $A$  restituisce 1  $\Leftrightarrow (x, 1) \in Q$
- $A$  ha costo  $O(f(n))$  in tempo (risp. in spazio), dove  $n$  indica la dimensione dell'input

Ma perché ci limitiamo ai problemi decisionali?

Così facendo:

- Ci concentriamo sul "calcolo" tralasciando eventuale tempo "perso" per restituire la soluzione (in problemi decisionali si restituisce un semplice booleano, in tempo costante)
- Possiamo caratterizzare un *lower bound* alla complessità dei corrispondenti problemi di ricerca/ottimizzazione
  - Es1: "trovare un albero di copertura per il grafo  $x$ " (problema di ricerca)  
è almeno tanto difficile quanto il corrispondente problema decisionale "esiste un albero di copertura per il grafo  $x$ ?"
  - Es2: "trovare un minimo albero di copertura per il grafo  $x$ " (problema di ottimizzazione)  
è almeno tanto difficile quanto il corrispondente problema decisionale "esiste un MST di costo  $\leq k$  per il grafo  $x$ ?"

La classe  $P$  è la classe dei problemi risolvibili in tempo polinomiale nella dimensione  $n$  dell'istanza di ingresso

$$P = \bigcup_{c=0}^{\infty} Time(n^c)$$

La classe  $PSpace$  è la classe dei problemi risolvibili in spazio polinomiale nella dimensione  $n$  dell'istanza di ingresso

$$PSpace = \bigcup_{c=0}^{\infty} Space(n^c)$$

La classe  $ExpTime$  è la classe dei problemi risolvibili in tempo esponenziale nella dimensione  $n$  dell'istanza di ingresso

$$ExpTime = \bigcup_{c=0}^{\infty} Time(2^{n^c})$$

Un algoritmo che richiede tempo polinomiale riuscirà al più ad accedere a un numero polinomiale di locazioni di memoria diverse, quindi

$$P \subseteq PSpace$$

Poiché  $n^c$  locazioni di memoria possono trovarsi al più in  $2^{n^c}$  stati diversi, si ha anche

$$PSpace \subseteq ExpTime$$

Non si sa se le inclusioni di sopra sono strette (ovvero non si sa se  $P \subset PSpace$  o se  $PSpace \subset ExpTime$ ) ma sicuramente almeno una delle due inclusioni è stretta! (poiché si sa che  $P \subset ExpTime$ ).

### Esempio:

Ogni espressione booleana si può trasformare in una *forma normale congiuntiva* (ovvero una congiunzione di clausole, dove una clausola è una disgiunzione di letterali, e un letterale è una variabile o una var. negata.

- Es:  $(x \vee \bar{y} \vee z) \wedge (y \vee z) \wedge (\bar{x} \vee y \vee z)$

Data una espressione booleana in forma normale congiuntiva, il problema della soddisfacibilità (*SAT*) richiede di verificare se esiste un'assegnazione di valori alle variabili che rende vero il predicato.

$$SAT \in PSpace \Rightarrow SAT \in ExpTime$$

- Se ci sono  $n$  variabili, basta tentare tutte le  $2^n$  assegnazioni
- Questo si fa con un algoritmo che richiede spazio lineare

Nei problemi di decisione siamo interessati a sapere se una certa istanza  $x$  *verifica una certa proprietà*

- Es: L'espressione booleana in forma normale congiuntiva  $k$  è soddisfacibile?

Spesso però siamo anche interessati a conoscere un qualche oggetto  $y$ , che dipende da  $x$  e dal problema da risolvere, che possa *certificare* il fatto che  $x$  gode di tale proprietà

- Es: un'assegnazione di variabili che rende l'espressione booleana  $k$  vera.

### Esempio:

Il problema delle formule booleane quantificate (tutte le variabili sono quantificate esistenzialmente o universalmente) richiede di verificare se una certa formula booleana quantificata è vera.

- Es:  $\exists x \forall y \exists z \forall w : (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee w) \wedge y$

Per il problema delle formule booleane quantificate **non** conosciamo certificati di dimensione polinomiale

- Quando l'espressione è vera, se le variabili quantificate universalmente sono  $k$  possiamo fornire un numero *esponenziale* ( $2^k$ ) di possibili assegnamenti per certificare che l'espressione è vera

## 11.2 NP

Informalmente  $NP$  è la classe dei problemi decisionali che ammettono certificati verificabili in tempo polinomiale

- Più precisamente, dato un problema  $Q \in NP$  esiste un algoritmo decisionale polinomiale di verifica  $D$  tale che:
  - Per ogni istanza  $x$   
Esiste un certificato di dimensione polinomiale  $C_x$  tale che  $D(C_x) = 1 \Leftrightarrow (x, 1) \in Q$
- Algoritmo “forza bruta” (*PSpace*) per problemi  $NP$ 
  - Genero tutti i possibili certificati polinomiali  $C$ , e controllo se  $D(C) = 1$
  - Problema: la quantità di possibili certificati solitamente non è polinomiale
- Sarebbe bello avere aiuto da un “*indovino*” che ci indica, se esiste, quale è il certificato giusto da verificare.

### Non-determinismo

Negli algoritmi visti ogni passo è sempre univocamente determinato dallo stato delle variabili

Un algoritmo decisionale **non deterministico**, invece, oltre alle normali istruzioni può eseguire istruzioni del tipo “*indovina*  $z \in S$ ”

- In altre parole, può indovinare un valore “corretto” e far proseguire la computazione nella “giusta” direzione
- Il non-determinismo è il nostro *indovino*

Proviamo ora a risolvere  $SAT$  usando il **non-determinismo**, si può fare in tempo lineare.

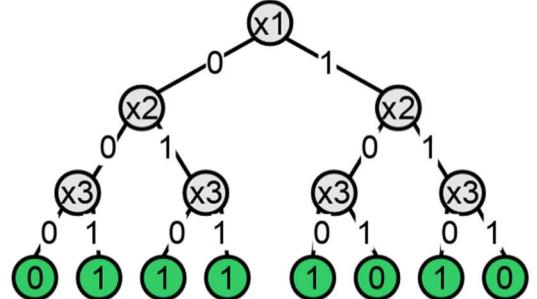
- Si esegue una sequenza di istruzioni *indovina*  $x_i \in \{0,1\}$  per ogni variabile  $x_i$  e poi si controlla che l'espressione sia vera.

**Esempio:** L'espressione seguente è soddisfacibile?

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

Nell'esempio abbiamo visto un algoritmo che può essere rappresentato da un albero di decisione

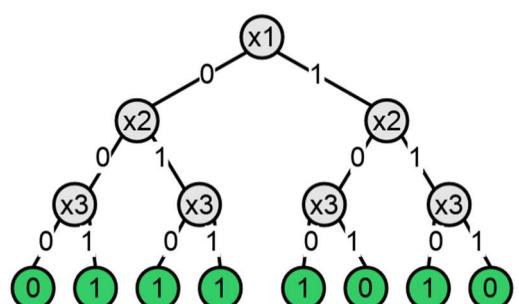
- L'algoritmo restituisce 1 se c'è almeno una foglia che restituisce 1, altrimenti restituisce 0
- Data un'istanza  $x$ , se esiste una sequenza di scelte nell'albero decisionale che porta ad una foglia che restituisce 1, tale sequenza può essere considerata un certificato  $C_x$ 
  - $C_x$  può essere verificato eseguendo l'algoritmo non deterministico in modo deterministico considerando le scelte in  $C_x$



Ma non sempre funziona;

Questo modo di procedere tramite “*indovini*” non funziona per l'espressione booleana quantificata:

$$\forall x_1 \forall x_2 (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

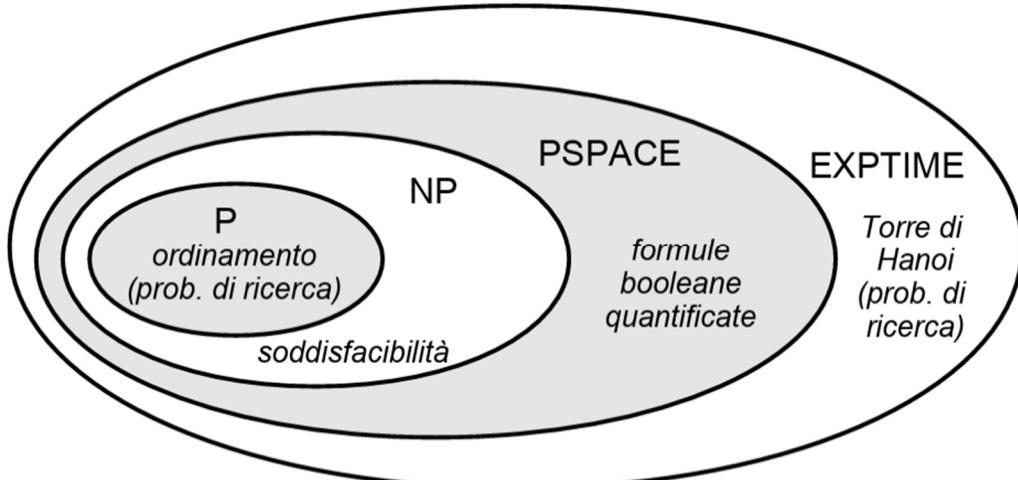


Data una funzione  $f(n)$ , chiamiamo  $NTime(f(n))$  l'insieme dei problemi decisionali che possono essere risolti da un algoritmo non deterministico in tempo  $O(f(n))$ .

La classe  $NP$  è la classe dei problemi risolvibili in tempo polinomiale non deterministico nella dimensione  $n$  dell'istanza d'ingresso:

$$NP = \bigcup_{c=0}^{\infty} NTime(n^c)$$

Quindi, riassumendo la gerarchia:



Dalle inclusioni qui sotto, almeno una è propria:

$$P \subseteq NP \subseteq PSpace \subseteq ExpTime$$

Si **congettura** che le inclusioni siano tutte proprie

### 11.2.1 Riducibilità polinomiale:

Consideriamo due problemi decisionali

- $Q_1 \subseteq l_1 \times \{0,1\}$
- $Q_2 \subseteq l_2 \times \{0,1\}$

Supponiamo di avere una funzione  $f: l_1 \rightarrow l_2$  in grado di trasformare in tempo polinomiale istanze di input per  $Q_1$  in istanze di input per  $Q_2$ , tali che per ogni soluzione  $s$

$$(x, s) \in Q_1 \Leftrightarrow (f(x), s) \in Q_2$$

Allora diremo che:

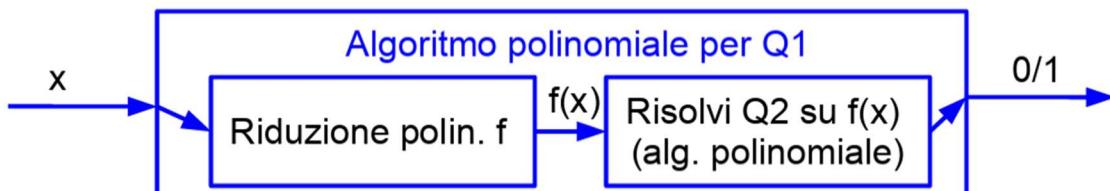
$Q_1$  è riducibile polinomialmente a  $Q_2$

$$Q_1 \leq_p Q_2$$

Consideriamo un problema  $Q_2$  per il quale sia noto un algoritmo risolutivo polinomiale  $\Rightarrow Q_2 \in P$

Supponiamo che  $Q_1$  sia riducibile polinomialmente a  $Q_2$ ; Allora anche  $Q_1 \in P$

- Per risolvere  $Q_1$  su istanza  $x$  basta trasformare in tempo polinomiale  $x$  nella relativa istanza  $f(x)$  per  $Q_2$ , e poi usare l'algoritmo risolutivo (polinomiale) che risolve  $Q_2$



### 11.2.2 NP-completezza

Un problema decisionale  $Q$  si dice **NP-arduo** se ogni problema  $W \in NP$  è riducibile polinomialmente a  $Q$

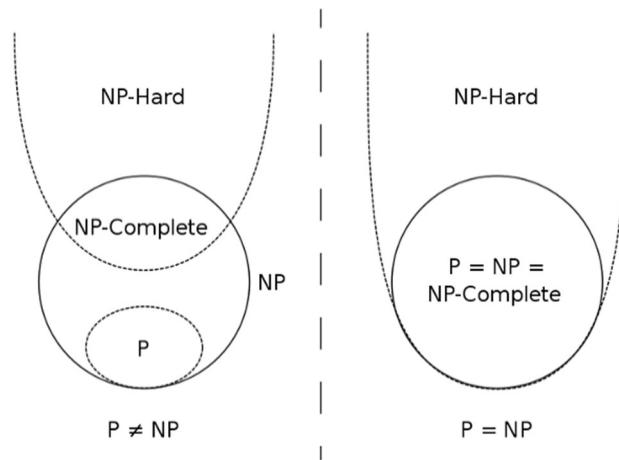
Un problema decisionale  $Q$  si dice **NP-completo** se appartiene alla classe  $NP$  ed è NP-hard.

#### Nota:

Se un qualunque problema decisionale NP-completo appartenesse alla classe  $P \Rightarrow P = NP$

- Sarebbe un disastro!
  - Il problema della *decifratura* di un documento crittografato sarebbe polinomiale (quindi eseguibile in tempo “ragionevoli”)
  - Infatti, se l’algoritmo di cifratura (polinomiale con chiave di cifratura) è noto, allora esiste un certificato polinomiale per il problema della decifratura: la password di cifratura!

#### Graficamente



#### Esempio1:

Problema della fermata limitata su istanza  $(X, k)$

**HALT**

- Dato il programma  $X$  ed un intero  $k$ , restituisce 1 se esiste un input per  $X$  che esegue al più  $k$  operazioni (0 altrimenti)

La fermata limitata è in  $NP$

- Dato un programma  $X$  e un intero  $k$ , il certificato è l’eventuale input  $y$  che termina in al più  $k$  passi (basta eseguire al più  $k$  passi di  $X$  sull’input  $y$ )

La fermata limitata è un problema NP-Hard

- Consideriamo un problema  $Q \in NP$  ed una sua istanza  $x$
- Riduco  $x$  in una istanza  $(X, k)$  del problema della fermata limitata:
  - Il programma  $X$  considera l’input come un certificato per istanza  $x$  di  $Q$ , se il certificato fallisce va in loop infinito, altrimenti termina.
  - $k = p(|x|)$  ( $p$  polinomio che indica il tempo di verifica dei certificati di  $Q$ )
- Si ha che  $(x, 1) \in Q \Leftrightarrow X$  ha fermata limitata in  $p(|x|)$  passi

#### Teorema di Cook

Il problema SAT è NP-completo

- La dimostrazione si basa su una riduzione del problema **HALT** in una espressione booleana in forma normale congiuntiva di dimensione polinomiale

#### Corollario:

Dato un problema  $Q$ , se esiste una riduzione polinomiale di **SAT** in  $Q$ , allora possiamo concludere che  $Q$  è un problema NP-hard.

**Esempio2:**

Problema della clique di dimensione  $k$ :

- Dato un grafo  $G$  verificare se esiste un sottografo completo di  $k$  vertici  
(un arco per ogni coppia di vertici)

Il problema è in NP: il certificato è l'insieme dei  $k$  vertici

Il problema è NP-hard: si riduce a SAT nel problema della clique

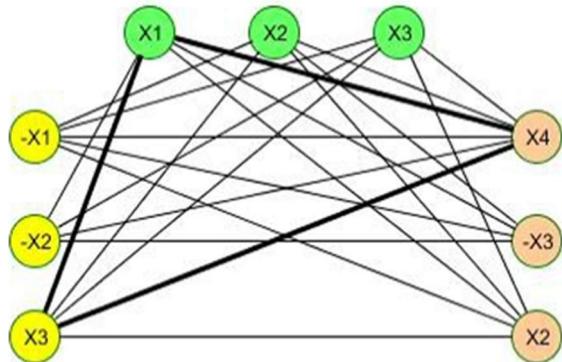
- Ogni clausola genera *un vertice per ogni suo letterale*
- Due vertici di due diverse clausole sono collegati da *un arco se non sono letterali incompatibili*
  - **non** sono la stessa variabile, una negata una no
- L'espressione è soddisfacibile  $\Leftrightarrow$  il grafo contiene una *clique di dimensione pari al numero di clausole*

Un esempio di riduzione da SAT a clique

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_4 \vee \overline{x_3} \vee x_2)$$

La clique indica anche validi assegnamenti:

$$x_1 = T \quad x_2 = T/F \quad x_3 = T \quad x_4 = T$$



Ci sono altri problemi NP-completi, sono mostrati e trattati meglio  
negli appunti e nel corso di Informatica Teorica.

**Conclusioni:**

I problemi NP-completi rappresentano il confine fra i problemi trattabili (risolvibili in tempo "ragionevoli" anche su input di dimensioni non banali) e i problemi non trattabili.

Molti problemi di interesse pratico sono NP-completi

Per questo motivo c'è forte interesse nello studio di questa classe di problemi