

Dati e algoritmi

0. Indice

- [0. Indice](#)
- [1. Nozioni](#)
 - [1.1. Problema computazionale](#)
 - [1.1.1. Esempi](#)
 - [1.1.2. Esercizi](#)
 - [1.2. Algoritmo e modello di calcolo](#)
 - [1.2.1. Algoritmo](#)
 - [1.2.2. Modello di calcolo RAM \(Random Access Machine\)](#)
 - [1.3. Pseudocodice](#)
 - [1.3.1. Esempio - Trasposta di una matrice \$n \times n\$ di interi](#)
 - [1.4. Taglia di un'istanza](#)
 - [1.4.1. Esempi](#)
 - [1.5. Struttura dati](#)
 - [1.5.1. Struttura dati - definizione](#)
 - [1.5.2. Esercizio](#)
 - [1.5.3. Esercizio](#)
- [2. Analisi degli algoritmi](#)
 - [2.1. Complessità in tempo](#)
 - [2.1.1. Requisiti per la complessità in tempo](#)
 - [2.1.2. Complessità \(in tempo\) al caso pessimo - definizione](#)
 - [2.1.3. Difficoltà nel calcolo di \$T_A\(n\)\$](#)
 - [2.1.4. Esempio \(arrayMax\)](#)
 - [2.2. Analisi asintotica](#)
 - [2.2.1. Esempio \(arrayMax\)](#)
 - [2.3. Ordini di grandezza](#)
 - [2.3.1. O-grande](#)
 - [2.3.2. Omega-grande](#)
 - [2.3.3. Theta](#)
 - [2.3.4. o-piccolo](#)
 - [2.3.5. Proprietà degli ordini di grandezza](#)
 - [2.3.6. Strumenti matematici](#)
 - [2.4. Analisi di complessità in pratica](#)

- [2.4.1. Limite superiore \(upper bound\) - definizione](#)
 - [2.4.2. Limite inferiore \(lower bound\) - definizione](#)
 - [2.4.3. Limiti superiori e inferiori](#)
 - [2.4.4. Terminologia per complessità](#)
 - [2.4.5. Esempio \(prefix averages\)](#)
 - [2.4.6. Esercizi](#)
 - [2.4.7. Regola di buon senso](#)
 - [2.4.8. Efficienza asintotica degli algoritmi](#)
 - [2.5. Analisi di correttezza](#)
 - [2.5.1. Induzione](#)
 - [2.5.2. Correttezza](#)
 - [2.5.3. Invariante](#)
 - [2.5.4. Correttezza di un ciclo tramite invariante](#)
 - [2.5.6. Ricorsione](#)
 - [2.5.7. Complessità di algoritmi ricorsivi](#)
 - [2.5.8. Correttezza di algoritmi ricorsivi](#)
 - [3. Ripasso di Java](#)
 - [3.1. Caratteristiche di Java e dell'approccio Object-Oriented](#)
 - [3.1.1. Modularità](#)
 - [3.1.2. Astrazione e incapsulamento \(information hiding\)](#)
 - [3.1.3. Ereditarietà \(inheritance\)](#)
 - [3.2. Programmazione generica \(generics\)](#)
 - [3.3. ADT elementari](#)
 - [3.3.1. Lista \(list\)](#)
 - [3.3.2. Pila \(stack\) e coda \(queue\)](#)
 - [3.4. Collection framework di Java](#)
 - [4. Alberi](#)
 - [4.1. Alberi generali](#)
 - [4.1.1. Definizioni e proposizioni](#)
-

[Lezione 01](#)

1. Nozioni

1.1. Problema computazionale

Un problema computazionale è costituito da

- un insieme I di *istanze* (i *possibili input*)

- un insieme S di *soluzioni* (i *possibili output*)
- una relazione Π che a ogni istanza $i \in I$ associa *una o più* soluzioni $s \in S$

❗ Osservazione

Π è un sottoinsieme del prodotto cartesiano $I \times S$

1.1.1. Esempi

Somma di Interi (\mathbb{Z})

- $\mathcal{I} = \{(x, y) : x, y \in \mathbb{Z}\};$
- $\mathcal{S} = \mathbb{Z};$
- $\Pi = \{((x, y), s) : (x, y) \in \mathcal{I}, s \in \mathcal{S}, s = x + y\}.$
Ad es: $((1, 9), 10) \in \Pi; \quad ((23, 6), 29) \in \Pi \quad ((13, 45), 31) \notin \Pi$

Ordinamento di array di interi

- $\mathcal{I} = \{A : A = \text{array di interi}\};$
- $\mathcal{S} = \{B : B = \text{array ordinati di interi}\};$
- $\Pi = \{(A, B) : A \in \mathcal{I}, B \in \mathcal{S}, B \text{ contiene gli stessi interi di } A\}.$
Ad es. $(\langle 43, 16, 75, 2 \rangle, \langle 2, 16, 43, 75 \rangle) \in \Pi$
 $(\langle 7, 1, 7, 3, 3, 5 \rangle, \langle 1, 3, 3, 5, 7, 7 \rangle) \in \Pi$
 $(\langle 13, 4, 25, 17 \rangle, \langle 11, 27, 33, 68 \rangle) \in \Pi$

Ordinamento di array di interi (ver.2)

- $\mathcal{I} = \{A : A = \text{array di interi}\};$
- $\mathcal{S} = \{P : P = \text{permutazioni}\};$
- $\Pi = \{(A, P) : A \in \mathcal{I}, P \in \mathcal{S}, P \text{ ordina gli interi di } A\}.$
Ad es. $(\langle 43, 16, 75, 2 \rangle, \langle 4, 2, 1, 3 \rangle) \in \Pi$
 $(\langle 7, 1, 7, 3, 3, 5 \rangle, \langle 2, 4, 5, 6, 1, 3 \rangle) \in \Pi$
 $(\langle 7, 1, 7, 3, 3, 5 \rangle, \langle 2, 5, 4, 6, 1, 3 \rangle) \in \Pi$
 $(\langle 13, 4, 25, 17 \rangle, \langle 1, 2, 4, 3 \rangle) \in \Pi$

❗ Osservazione

- Istanze diverse possono avere la stessa soluzione (come la somma)
- Un'istanza può avere diverse soluzioni (come l'ordinamento ver. 2)

1.1.2. Esercizi

Esercizio

Specificare come problema computazionale Π la verifica se due insiemi finiti di oggetti da un universo U sono disgiunti oppure no.

$I \equiv \{(A, B) : A, B \subseteq U, A, B \text{ finiti}\}$

$S \equiv \{true, false\}$

$\Pi \equiv \{((A, B), s) \text{ se } A \cap B = \emptyset \text{ allora } s = true, \text{ se } A \cap B \neq \emptyset \text{ allora } s = false\} \quad s \in S, (A, B) \in I$

Esercizio

Specificare come problema computazionale Π la ricerca dell'inizio e della lunghezza del più lungo segmento di 1 consecutivi in una stringa binaria.

$I \equiv \{A : A \text{ è una stringa binaria}\}$

$S \equiv \{(i, l) : i, l \in \mathbb{N}_0\}$

$\Pi \equiv \{(A, (i, l)) : i \text{ la casella di inizio del segmento di 1 consecutivi più numeroso, } l \text{ la lunghezza del segmento di 1 consecutivi più lungo}\}$

1.2. Algoritmo e modello di calcolo

1.2.1. Algoritmo

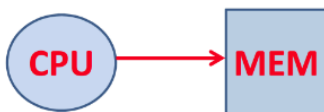
Un *algoritmo* procedura computazionale ben definita che trasforma un dato *input* in un *output* eseguendo una sequenza finita di *operazioni elementari*.

□

L'algoritmo fa riferimento a un *modello di calcolo*, ovvero un'astrazione di computer che definisce l'insieme di operazioni elementari.

Le operazioni elementari sono: assegnamento, operazioni logiche, operazioni aritmetiche, indicizzazione di array, return di un valore da parte di un metodo, ecc.

1.2.2. Modello di calcolo RAM (Random Access Machine)



In questo modello input, output, dati intermedi (e il programma) si trovano in memoria.

Un algoritmo A risolve un *problema computazionale* $\Pi \subseteq I \times S$ se:

1. A calcola una funzione da I a S e quindi,
 - riceve come input istanze $i \in I$

- produce come output soluzioni $s \in S$
2. Dato $i \in I$, A produce in output $s \in S$ tale che $(i, s) \in \Pi$.
 Se Π associa più soluzioni a una istanza i , per tale istanza A ne calcola una (quale dipende da come è stato progettato).

Lezione 02

1.3. Pseudocodice

Per semplicità e facilità di analisi, descriviamo un algoritmo utilizzando uno pseudocodice strutturato come segue:

Algoritmo nome(parametri)

Input: breve descrizione dell'istanza di input

Output: breve descrizione della soluzione restituita in output
 Descrizione chiara dell'algoritmo tramite costrutti di linguaggi di programmazione e, se utile ai fini della chiarezza, anche tramite linguaggio naturale, dalla quale sia facilmente determinabile la sequenza di operazioni elementari eseguita per ogni dato input.

"Algoritmo nome (parametri)" è la firma (signature) dell'algoritmo; i parametri sono l'input.

Si può usare il linguaggio naturale per cose semplici, che sarebbero lunghe da scrivere con i costrutti del linguaggio di programmazione.

Usare sempre questa struttura per lo pseudocodice!

Per maggiori dettagli fare riferimento alla dispensa sulla [Specificazione di Algoritmi in Pseudocodice](#), disponibile sul Moodle del corso.

1.3.1. Esempio - Trasposta di una matrice $n \times n$ di interi

Problema computazionale: $\Pi \subseteq I \times S$

$I \equiv \{A: \text{matrice } n \times n \text{ di interi}\}$

$S \equiv \{B: \text{matrice } n \times n \text{ di interi}\}$

$\Pi \equiv \{(A, B) : A \in I, B \in S, B = A^t\}$

Algoritmo: idea per $n = 4$

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

Scambio ciascuna entry a_{ij} della matrice triangolare superiore ($a_{ij} : i = 0, \dots, n-2, j > i$) con a_{ji} (l'omologa del triangolo inferiore)

Algoritmo transpose(A)

Input: matrice A $n \times n$ di interi a_{ij} , $0 \leq i, j \leq n$.

Output: matrice A^t .

```
for i <- 0 to n-2 do{
    for j <- i+1 to n-1 do{
        scambia a_ij con a_ji
    }
}
return A
```

1.4. Taglia di un'istanza

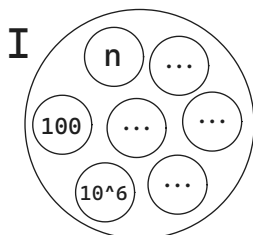
L'analisi di un algoritmo (ad esempio la determinazione della sua complessità) viene solitamente fatta *partizionando le istanze in gruppi in base alla loro taglia (size)*, in modo che le istanze di un gruppo siano tra loro *confrontabili*.

La *taglia di un'istanza* è espressa da uno o più valori che ne caratterizzano la grandezza.

1.4.1. Esempi

- Se ho n elementi da ordinare con un MergeSort, la taglia è n . In questo caso fa riferimento alla dimensione fisica dell'istanza.

La taglia serve per dividere in gruppi omogenei da analizzare separatamente.



- Ordinamento di un array:
Taglia n = numero di elementi dell'array
- Trasposta di una matrice:
Prendendo una generica matrice $n \times m$ ho più modi di definire la

taglia, in base al tipo di analisi che si vuole condurre:

- Taglia $x = n * m$
- Taglia (n, m) , con n e m rispettivamente il numero di righe e numero di colonne
- (Se $n = m$) Taglia $n =$ numero di righe/colonne

1.5. Struttura dati

Le strutture dati sono usate dagli algoritmi per organizzare e accedere in modo sistematico ai dati di input e ai dati generati durante l'esecuzione.

1.5.1. Struttura dati - definizione

Una *struttura dati* è una collezione di oggetti corredata di *metodi* di accesso e/o modifica.

Vi sono due *livelli di astrazione*:

1. *Livello logico*: specifica l'organizzazione logica degli oggetti della collezione, e la relazione input-output di ciascun metodo (a questo livello si parla di *Abstract Data Type* o ADT)
2. *Livello fisico*: specifica il layout fisico dei dati e la realizzazione dei metodi tramite algoritmi.

1.5.1.1. Esempio

In Java a livello logico ho (gerarchia di) *interfacce*, mentre a livello fisico ho (gerarchia di) *classi*.

□

1.5.2. Esercizio

Siano A e B due array di n e m interi, rispettivamente. Scrivere un algoritmo in pseudocodice per calcolare il seguente valore:

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (A[i] \cdot B[j])$$

1.5.3. Esercizio

Sia A un array di n interi distinti, ordinato in senso crescente, e x un valore intero. Scrivere due algoritmi in pseudocodice che implementano la ricerca binaria per trovare x in A . Entrambi gli algoritmi restituiscono l'indice i tale che $A[i] = x$, se x appare in

A, e -1 altrimenti. Il primo algoritmo deve essere iterativo e usare un solo ciclo, e il secondo algoritmo deve essere ricorsivo.

2. Analisi degli algoritmi

L'analisi di un algoritmo A mira a studiarne l'*efficienza* e l'*efficacia*. In particolare, essa può valutare la *complessità* di *tempo* e *spazio*, e la *correttezza* di *terminazione* (se termina o rimane in un loop) e della *soluzione del problema computazionale*.

Noi ci concentreremo sulla complessità di tempo e sulla correttezza della soluzione del problema computazionale.

2.1. Complessità in tempo

L'*obiettivo* è *stimare il tempo di esecuzione ("running time") di un algoritmo* al fine di valutarne l'efficienza e poterlo confrontare con altri algoritmi per lo stesso problema.

2.1.1. Requisiti per la complessità in tempo

La complessità in tempo deve:

1. Riguardare *tutti gli input*;
2. Permettere di *confrontare algoritmi* (senza necessariamente determinare il tempo di esecuzione esatto);
3. Essere *derivabile dallo pseudocodice*.

2.1.1.1. Esempio

Stimare sperimentalmente il tempo di esecuzione di un algoritmo (ad esempio in Java con `System.currentTimeMillis()`) è utile, ma non soddisfa i requisiti. **Perché?**

- Non si possono considerare tutti gli input (se infiniti);
- Richiede di implementare l'algoritmo con un programma e l'impatto dell'implementazione può influenzare il confronto tra algoritmi;
- La stima dipende dall'ambiente hardware/software.

2.1.1.2. Approccio che soddisfa i requisiti

- Analisi al *caso pessimo* (*worst-case*) in funzione della taglia dell'istanza (requisiti 1 e 2)
- Conteggio delle operazioni elementari nel modello RAM (requisiti 2 e 3)
- Analisi asintotica (per semplificare il conteggio)

! Osservazione

Esiste anche l'analisi al caso medio (*average case*) e l'analisi probabilistica.

2.1.2. Complessità (in tempo) al caso pessimo - definizione

Sia A un algoritmo che risolve $\Pi \subseteq I \times S$.

La complessità (in tempo) al caso pessimo di A è una funzione $t_A(n)$ definita come *il massimo numero di operazioni elementari che A esegue per risolvere un'istanza di taglia n* .

In altre parole, se chiamiamo $t_{A,i}$ il numero di operazioni eseguite da A per l'istanza i , abbiamo che

$$t_A(n) = \max\{t_{A,i} : \text{istanza } i \in I \text{ di taglia } n\}$$

La definizione rispetta i tre requisiti definiti sopra.

2.1.3. Difficoltà nel calcolo di $t_A(n)$

Determinare $t_A(n)$ per ogni n è arduo, se non impossibile, perché è *difficile identificare l'istanza peggiore di taglia n* e perché è *difficile contare il massimo numero di operazioni richieste per risolvere tale istanza* peggiore (il conteggio richiederebbe anche una specifica dettagliata del set di operazioni elementari del modello RAM).

Ma *non è necessario determinare esattamente $t_A(n)$* , infatti il tempo di esecuzione, che la complessità vuole stimare, dipende da tanti fattori che è impossibile quantificare in modo preciso, in più le diverse operazioni elementari del modello RAM possono avere impatto diverso sui tempi di esecuzione a seconda delle architetture.

Ci accontentiamo dei *limiti superiori* e i *limiti inferiori* a $t_A(n)$.

2.1.4. Esempio (arrayMax)

Algoritmo arrayMax(A)

Input: array $A[0 \div n-1]$ di $n \geq 1$ interi.

Output: max intero in A .

```
currMax ← A[0]
for i ← 1 to n-1 do{
    if (currMax < A[i]) then currMax ← A[i];
```

```
}  
return currMax
```

Taglia dell'istanza: n (ragionevole)

2.1.4.1. Stima di $t_{\text{arrayMax}}(n)$

È facile vedere che per una qualsiasi istanza di taglia n :

- al di fuori del ciclo `for arrayMax` esegue un numero costante (rispetto a n) di operazioni;
- in ciascuna delle $n-1$ iterazioni del ciclo `for` si esegue un numero costante di operazioni.

Esistono allora quattro costanti $c_1, c_2, c_3, c_4 > 0$ tali che per ogni n valga $t_{\text{arrayMax}}(n) \leq c_1 n + c_2$ (cioè il *limite superiore*) e $t_{\text{arrayMax}}(n) \geq c_3 n + c_4$ (cioè il *limite inferiore*). Non è necessario stimare le quattro costanti per le stesse ragioni per cui non è necessario stimare esattamente la complessità.

2.2. Analisi asintotica

Si ricorre quindi all'analisi asintotica, ignorando i *fattori moltiplicativi costanti* (rispetto alla taglia dell'istanza) e i *termini additivi non dominanti*.

2.2.1. Esempio (`arrayMax`)

Riprendendo l'esempio, nel caso di `arrayMax` possiamo affermare che $t_{\text{arrayMax}}(n)$ è *al più* proporzionale a n (limite superiore) ed è anche *almeno* proporzionale a n (limite inferiore), ne consegue quindi che $t_{\text{arrayMax}}(n)$ è *proporzionale* a n (limite stretto).

□

Per esprimere affermazione come quelle fatte sopra si usano gli **ordini di grandezza**: $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$.

[Lezione 03](#)

2.3. Ordini di grandezza

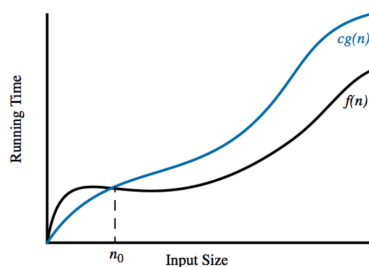
Siano $f(n)$, $g(n)$ funzioni da \mathbb{N} a $\mathbb{R}^+ \cup \{0\}$.

2.3.1. O-grande

$f(n) \in O(g(n))$ se $\exists c > 0$ e $\exists n_0 \geq 1$, *costanti rispetto a n* , tali che

$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

Cioè se $f(n)$ è al più proporzionale a $g(n)$, ovvero se $f(n)$ non cresce asintoticamente più di $c \cdot g(n)$.



2.3.1.1. Esempi

$f(n)$	$O(\cdot)$	c	n_0
$3n + 4$ per $n \geq 1$	$O(n)$	4	4
$n + 2n^2$ per $n \geq 1$	$O(n^2)$	3	1
2^{100} per $n \geq 1$	$O(1)$	2^{100}	1
$c_1n + c_2$ per $n \geq 1, c_1, c_2 > 0$	$O(n)$	$c_1 + c_2$	1

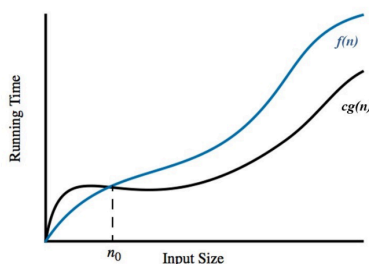
Si può dire che $3n + 4 \in O(n^5)$, $c = 7$, $n_0 = 1$, ma non sarebbe molto utile.

2.3.2. Omega-grande

$f(n) \in \Omega(g(n))$ se $\exists c > 0$ e $\exists n_0 \geq 1$, *costanti rispetto a n* , tali che

$$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

Cioè $f(n)$ è almeno proporzionale a $g(n)$.



2.3.2.1. Esempi

$f(n)$	$\Omega(\cdot)$	c	n_0
$3n + 4$ per $n \geq 1$	$\Omega(n)$	1	4
$n + 2n^2$ per $n \geq 1$	$\Omega(n^2)$	1	1

$f(n)$	$\Omega(\cdot)$	c	n_0
2^{100} per $n \geq 1$	$\Omega(1)$	2^{100}	1
$c_1 n + c_2$ per $n \geq 1, c_1, c_2 > 0$	$\Omega(n)$	1	1

2.3.3. Theta

$f(n) \in \Theta(g(n))$ se

$$f(n) \in O(g(n)) \text{ e } f(n) \in \Omega(g(n))$$

Cioè $f(n)$ è (*esattamente*) proporzionale a $g(n)$.

2.3.3.1. Esempi

- $f(n) = 3n + 4 \in \Theta(n)$;
- $f(n) = n + 2n^2 \in \Theta(n^2)$;
- $f(n) = 2^{100} \in \Theta(1)$;
- $t_{\text{arrayMax}}(n) \in \Theta(n)$.

2.3.4. o-piccolo

$f(n) \in o(g(n))$ se

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

Cioè $f(n)$ è *asintoticamente* più piccola (cresce meno) di $g(n)$.

2.3.4.1. Esempi

- $f(n) = 100n$ per $n \geq 1 \implies f(n) \in o(n^2)$;
- $f(n) = \frac{3n}{\log_2 n}$ per $n \geq 1 \implies f(n) \in o(n)$.

2.3.5. Proprietà degli ordini di grandezza

1. $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$ per ogni $f(n), g(n) : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$;
2. $\sum_{i=0}^k a_i n^i \in \Theta(n^k)$, se $a_k > 0$, k, a_i costanti, e $k \geq 0$.
Ad esempio: $(n+1)^5 \in \Theta(n^5)$.
Cioè tengo il termine con l'esponente maggiore;
3. $\log_b n \in \Theta(\log_a n)$, se $a, b > 1$ sono costanti;

⚠ Osservazione

La proprietà deriva dalla relazione $\log_b n = (\log_a n)(\log_b a)$ e, grazie a essa, *la base dei logaritmi*, se costante, *si omette*

negli ordini di grandezza, a meno che il logaritmo non sia all'esponente.

4. $n^k \in o(a^n)$, se $k > 0$, $a > 1$ sono costanti;
5. $(\log_b n)^k \in o(n^h)$ se b, k, h sono costanti, con $b > 1$ e $h, k > 0$.

2.3.6. Strumenti matematici

2.3.6.1. Parte bassa

$\forall x \in \mathbb{R}$, si definisce $\lfloor x \rfloor$ come il più grande intero tale che sia $\leq x$.

2.3.6.1.1. Esempi

- $\lfloor \frac{3}{2} \rfloor = 1$;
- $\lfloor 3 \rfloor = 3$.

2.3.6.2. Parte alta

$\forall x \in \mathbb{R}$, si definisce $\lceil x \rceil$ come il più piccolo intero tale che sia $\geq x$.

2.3.6.2.1. Esempi

- $\lceil \frac{3}{2} \rceil = 2$;
- $\lceil 3 \rceil = 3$.

2.3.6.3. Modulo

$\forall x, y \in \mathbb{Z}$, con $y \neq 0$, si definisce $x \bmod y$ come il resto della divisione intera x/y (l'operatore "%" in Java).

2.3.6.3.1. Esempi

- $29 \bmod 7 = 4$;
- $80 \bmod 4 = 0$.

2.3.6.4. Sommatorie notevoli

$\forall n \in \mathbb{Z}$ e $a \in \mathbb{R}$, con $n \geq 0$ e $a > 0$ vale:

- $\sum_{i=0}^n i = \frac{n(n+1)}{2} \in \Theta(n^2)$;
- $\sum_{i=0}^n a^i = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$ per $a > 1$;
- $\sum_{i=1}^n a^i = \frac{a^{n+1}-1}{a-1} - 1 \in \Theta(a^n)$ per $a > 1$.

2.4. Analisi di complessità in pratica

Dato un algoritmo A e detta $t_A(n)$ la sua complessità al caso pessimo, si cercano limiti asintotici superiori e/o inferiori a $t_A(n)$.

2.4.1. Limite superiore (upper bound) - definizione

$$t_A(n) \in O(f(n))$$

Si prova argomentando che per ogni n "abbastanza grande" e per ciascuna istanza di taglia n l'algoritmo esegue $\leq c \cdot f(n)$ operazioni, con c costante (e che non serve determinare).

2.4.2. Limite inferiore (lower bound) - definizione

$$t_A(n) \in \Omega(f(n))$$

Si prova argomentando che per ogni n "abbastanza grande", *esiste* un'istanza di taglia n per la quale l'algoritmo esegue $\geq c \cdot f(n)$ operazioni, con c costante (e che non serve determinare).

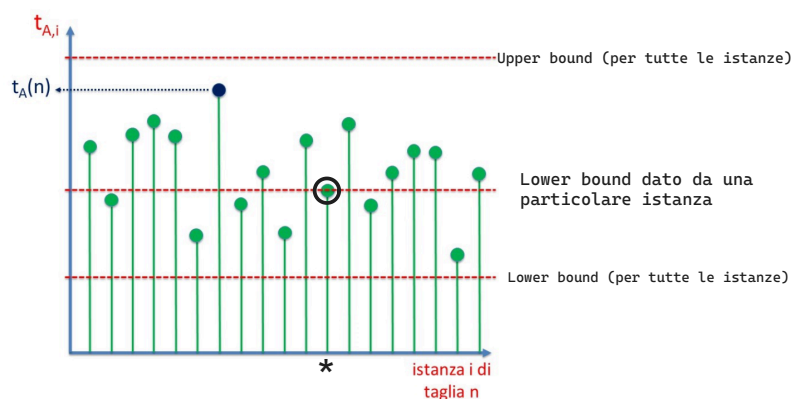
In alcuni casi è comodo argomentare che per *ciascuna* istanza di taglia n l'algoritmo esegue $\geq c \cdot f(n)$ operazioni.

⚠ Attenzione

Sia che si provi $t_A(n) \in O(f(n))$ o che si provi $t_A(n) \in \Omega(f(n))$

- $f(n)$ deve essere più vicino possibile alla complessità vera (*tight bound*)
- $f(n)$ deve essere più semplice possibile, quindi senza costanti e termini additivi di ordine inferiore, solo con i *termini essenziali*!

2.4.3. Limiti superiori e inferiori



2.4.4. Terminologia per complessità

- logaritmica: $\Theta(\log n)$, base 2 o costante > 1
- lineare: $\Theta(n)$
- quadratica: $\Theta(n^2)$
- cubica: $\Theta(n^3)$
- polinomiale $\Theta(n^c)$, $c > 0$ costante
- esponenziale: $\Omega(a^n)$, $a > 1$ costante
- polilogaritmica: $\Theta((\log n)^c)$, $c > 0$ costante

2.4.5. Esempio (prefix averages)

Si consideri il seguente problema computazionale.

Dato un array di n interi $X[0 \dots n-1]$ calcolare un array $A[0 \dots n-1]$ dove $A[i] = \left(\sum_{j=0}^i X[j]\right) \frac{1}{i+1}$, per $0 \leq i < n$.

Vedremo adesso due algoritmi di cui uno banale e inefficiente, poi uno più furbo ed efficiente. Per entrambi gli algoritmi vale la seguente specifica di input-output:

Input: $X[0 \dots n-1]$ array di n interi.

Output: $A[0 \dots n-1] : A[i] = \left(\sum_{j=0}^i X[j]\right) \frac{1}{i+1}$ per $0 \leq i < n$.

2.4.5.1. Algoritmo inefficiente

Algoritmo `prefixAverages1`

```
for i <- 0 to n-1 di{
  a <- 0;
  for j <- 0 to i do {
    a <- a+X[j];
  }
  A[i] <- a/(i+1);
}
return A
```

- Fuori dal for esterno: $\Theta(1)$ operazioni
- Per ciascuna iterazione i del for esterno ($i = 0, \dots, n-1$):
 - $\Theta(i+1)$ operazioni nel for interno
 - $\Theta(1)$ altre operazioni

La complessità è quindi:

$$t_{pA1}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1} i + 1\right) = \Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(\frac{(n-1)n}{2}\right) = \Theta(n^2).$$

2.4.5.2. Algoritmo efficiente

Algoritmo prefixAverages2

```
s <- 0;
for i <- 0 to n-1 do{
    s <- s + X[i];
    A[i] <- s/(i+1);
}
return A
```

- Fuori dal for: $\Theta(1)$ operazioni
- Iterazioni i del for ($i = 0, \dots, n-1$): $\Theta(1)$ operazioni

La complessità è quindi: $t_{pA2}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1} 1\right) = \Theta(n)$

Possiamo affermare che $t_{pA2}(n) \in o(t_{pA1}(n))$, cioè è migliore.

[Lezione 04](#)

2.4.6. Esercizi

2.4.6.1. Analisi complessità in tempo

Sia A un algoritmo che ricevuto in input un array X di $n \geq 1$ interi esegue

- $c_1 n$ operazioni per ogni intero pari in X
 - $c_2 \lceil \log_2 n \rceil$ operazioni per ogni intero dispari in X
- dove c_1 e c_2 sono costanti positive.

Analizzare la complessità in tempo dell'algoritmo esprimendola con $\Theta(\cdot)$.

Per un'istanza particolare (es: array di tutti interi pari) ho $\Theta(n^2)$, ho quindi ricavato un lower bound.

- Taglia dell'istanza: n
- Complessità: $t_A(n)$
- Upper bound: \forall istanza di taglia n , A esegue
 $\leq n \max\{c_1, c_2 \lceil \log_2 n \rceil\} \leq n \max\{c_1, c_2\} \max\{n, \log_2 n\} = n \max\{c_1, c_2\} n$ operazioni
 $\implies t_A(n) \in O(n^2)$ **(1)**
- Lower bound basato su una particolare istanza:
Sia X un array di interi pari \implies per tale X , A esegue
 $n \cdot c_1 \cdot n = c_1 n^2$ operazioni $\implies t_A(n) \in \Omega(n^2)$ **(2)**

- Lower bound basato su tutte le istanze:
 \forall istanza di taglia n , A esegue
 $\geq n \min\{c_1 n, c_2 \lceil \log_2 n \rceil\} \geq n \min\{c_1, c_2\} \min\{n, \log_2 n\} = n \min\{c_1, c_2\} \lceil \log_2 n \rceil$
operazioni $\implies t_A(n) \in \Omega(n \log n)$
Questo lower bound è peggiore rispetto a quello trovato prima.

2.4.6.1. Descrizione insertion sort

Il seguente pseudocodice descrive l'algoritmo di ordinamento chiamato `InsertionSort` (Si assume che la sequenza S da ordinare sia una variabile globale che dopo la fine dell'algoritmo è accessibile e ordinata).

Algoritmo `InsertionSort(S)`

Input: Sequenza $S[0 \dots n-1]$ di n chiavi.

Output: Sequenza S ordinata in senso crescente.

```
for i<-1 to n-1 do{
  curr<-S[i];
  j<-i-1;
  while((j>=0) AND (S[j] > curr)) do{
    S[j+1]<-S[j];
    j<-j-1;
  }
  S[j+1]<-curr;
}
```

Notazione

Nel caso degli algoritmi di ordinamento, in analogia al libro di testo, usiamo il termine *sequenza* per denotare la collezione di elementi da ordinare che assumiamo rappresentata come array.

1. Trovare opportune funzioni $f_1(n)$, $f_2(n)$, $f_3(n)$ tali che le seguenti affermazioni siano vere, per una qualche costante $c > 0$ e per n abbastanza grande.
 - Per ciascuna istanza di taglia n l'algoritmo esegue $\leq c f_1(n)$ operazioni.
 - Per ciascuna istanza di taglia n l'algoritmo esegue $c f_2(n)$ operazioni.
 - Esiste un'istanza di tagli a n per la quale l'algoritmo esegue $\geq c f_3(n)$ operazioni

La funzione $f_1(n)$ deve essere la più piccola possibile, mentre le funzioni $f_2(n)$ e $f_3(n)$ devono essere le più grandi possibili.

2. Sia $t_{IS}(n)$ la complessità al caso peggio dell'algoritmo.

Sfruttando le affermazioni del punto precedente trovare un upper bound $O(\cdot)$ e un lower bound $\Omega(\cdot)$ per $t_{IS}(n)$.

2.4.7. Regola di buon senso

Una complessità polinomiale (o migliore) implica un algoritmo efficiente, mentre una complessità esponenziale implica un algoritmo inefficiente.

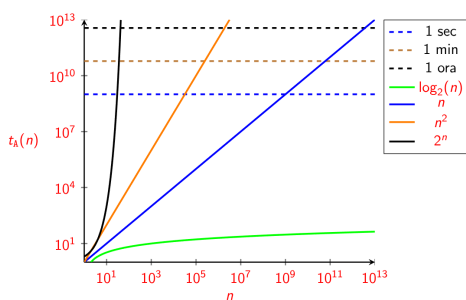
Supponiamo che la complessità $t_A(n)$ sia espressa in nanosecondi e definiamo

$$n_\tau \equiv \text{max taglia di un'istanza risolvibile in tempo } \tau$$

Per ottenere n_τ , risolviamo $t_A(n_\tau) = \tau$ rispetto a n_τ .

2.4.7.1. Esempi

$t_A(n)$	n_τ per $\tau = 10^9$ (1 sec)	n_τ per $\tau = 60 * 10^9$ (1 min)	n_τ per $\tau = 3600 * 10^9$ (1 ora)
$\log_2 n$	$2^{10^9} = \infty$	∞	∞
n	10^9	$6 * 10^{10}$	$3.6 * 10^{12}$
n^2	$10^{4.5}$	$\approx 8 * 10^{4.5}$	$\approx 1.8 * 10^6$
2^n	≈ 30	≈ 36	≈ 42



2.4.7.2. Esempio

La crittografia a chiave pubblica è molto usata dai protocolli di sicurezza.

L'invio di un messaggio cifrato da Alice a Bob funziona (in maniera approssimata) in questo modo:

- Bob possiede una chiave privata k_1 e una chiave pubblica k_2 ;
- Alice invia a Bob un messaggio m cifrato con k_2 ;

- Bob decifra il messaggio ricevuto da Alice con k_1 ;

L'*Algoritmo RSA* sviluppato nel 1977 da Rivest, Shamir, Adleman (*Turing Award 2002*) è il più famoso algoritmo di crittografia a chiave pubblica. La sua “sicurezza” si basa sulla difficoltà di risolvere il seguente problema.

2.4.7.2.1. Integer factorization (per interi prodotti di 2 primi)

Dato un intero N prodotto di due primi p, q , determinare p e q .

Che taglia dell'istanza scelgo?

Posso scrivere, ad esempio, questo algoritmo banale per risolvere il problema:

```
for p<-2 to floor{sqrt{N}} do{
    if(N mod p = 0) then return {p, N/p};
}
```


Esprimiamo la complessità in funzione del numero n bit che servono per rappresentare N . Se $p, q \in \Theta(\sqrt{N})$ la complessità è $\Theta(\sqrt{N}) = \Theta(2^{n/2})$

2.4.7.2.1.1. Esempio numerico

Prendendo $n = 1024$ ottengo $2^{n/2} = 2^{512} \geq 10^{154}$. Sul computer più potente del mondo ($< 10^{19}$ flop/sec) l'algoritmo banale richiederebbe circa $\frac{10^{154}}{10^{19}} = 10^{135}$ secondi. Considerando che un anno ha meno di 10^8 secondi, sarebbero più di 10^{127} anni di calcolo.

Nota bene

Esistono algoritmi più efficienti, ma sempre con complessità esponenziale.

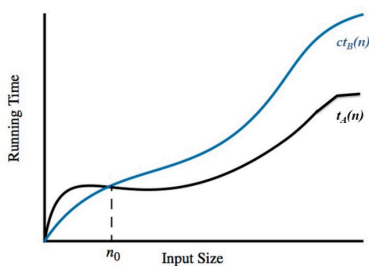
 Dalla motivazione del A.M. Turing Award 2002 assegnato a Rivest, Shamir e Adleman

RSA is used in almost all internet-based commercial transactions. Without it, commercial online activities would not be as widespread as they are today. It allows users to communicate sensitive information like credit card numbers over an unsecure internet without having to agree on a shared secret

key ahead of time. Most people ordering items over the internet don't know that the system is in use unless they notice the small padlock symbol in the corner of the screen. RSA is a prime example of an abstract elegant theory that has had great practical application.

2.4.8. Efficienza asintotica degli algoritmi

Dati A , B due algoritmi che risolvono il problema computazionale Π , $t_A(n)$ e $t_B(n)$ sono le complessità rispettivamente di A e B al caso pessimo. Se $t_A(n) \in o(t_B(n))$, allora A è "*asintoticamente più efficiente*" di B .



Nota bene: n_0 potrebbe essere *molto* grande.

💡 Caveat sull'analisi asintotica al caso pessimo

1. *Le costanti trascurate potrebbero essere elevate* e, in pratica, potrebbero avere un impatto elevato sulle prestazioni.
2. *Il caso pessimo potrebbe essere costituito solo da istanze patologiche* mentre per tutte le istanze di interesse la complessità potrebbe essere asintoticamente migliore.
Cosa fare in questo caso?
 - Restringere il dominio delle istanze, mantenendo quelle di interesse ed escludendo quelle patologiche.
 - Fare un'analisi al caso medio.

2.5. Analisi di correttezza

2.5.1. Induzione

Per provare che una proprietà $Q(n)$ è vera $\forall n \geq n_0$ si procede così:

- Si sceglie un intero $k \geq 0$;

- **Base:** si dimostra $Q(n_0), Q(n_0+1), \dots, Q(n_0+k)$;
- **Passo induttivo:** si fissa un valore $n \geq n_0+k$ *arbitrario* e si dimostra che $Q(m)$ vera $\forall m: n_0 \leq m \leq n \implies Q(n+1)$ vera.

⚠ Osservazioni

- " $Q(m)$ vera $\forall m: n_0 \leq m \leq n$ " è chiamata *ipotesi induttiva*.
- La dimostrazione deve valere *per ogni* $n \geq n_0+k$
- Di solito, ma non sempre, $k=0$ (il libro di testo descrive l'induzione con $n_0=1$).

2.5.1.1. Esempio

$$Q(n) : \sum_{i=0}^n i = \frac{n(n+1)}{2} \quad \forall n \geq 0$$

Nota bene: Implica che $Q(n) \in \Theta(n^2)$.

- $n_0 = 0, k = 0$
- Base: $Q(0) : \sum_{i=0}^0 i = 0 = \frac{0 \cdot 1}{2} \checkmark$
- Passo induttivo: Fisso $n \geq n_0 + n = 0$ arbitrario.
Ipotesi induttiva: $\sum_{i=0}^m i = \frac{m(m+1)}{2} \quad \forall 0 \leq m \leq n$
Dimostriamo che $Q(n+1)$ è vera:
 $\sum_{i=0}^{n+1} i = n+1 + \sum_{i=0}^n i = n+1 + \frac{n(n+1)}{2} =$ per ipotesi induttiva
 $= \frac{(n+1)(n+2)}{2} \implies Q(n+1)$ è vera

2.5.1.2. Esercizio

Dimostrare per induzione la seguente proprietà:

$$Q(n) : \sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad \forall n \geq 0$$

Nota bene: Implica che $\sum_{i=0}^n i^2 \in \Theta(n^3)$

⚠ Osservazione

In generale $\sum_{i=0}^n i^k \in \Theta(n^{k+1})$, con k costante.

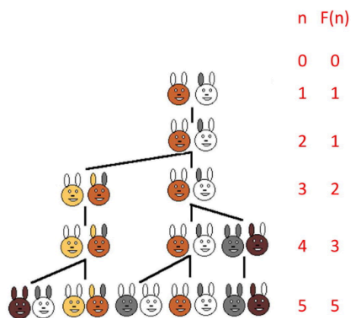
2.5.1.3. Successione di Fibonacci - esempio

La *successione di Fibonacci* è definita così:

$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n+1) = F(n) + F(n-1) \end{cases}$$

$\forall n \geq 1$.

Si può pensare a $F(n)$ come il numero di coppie di conigli all'inizio del mese n se una coppia genera un'altra coppia ogni mese, a partire dal terzo mese, i conigli non muoiono, all'inizio del mese 1 c'è una coppia neonata.



Dimostrare che

$$F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \quad (\implies F(n) \in \Theta(N\phi^n))$$

$\forall n \geq 0$, dove $\phi = \frac{1+\sqrt{5}}{2}$ (*golden ratio*) e $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

2.5.1.3.1. Induzione fallace

Dimostriamo $F(n) = 0, \forall n \geq 0$ ($n_0 = 0$):

- $k = 0$
- Base: $F(0) = 0 \implies \checkmark$
- Passo induttivo: fissato $n \geq 0$ e assumendo $F(m) = 0$ per ogni $0 \leq m \leq n$ (ipotesi induttiva), si ha $F(n+1) = F(n) + F(n-1) = 0 + 0$

Il passo induttivo non è corretto quando $n=0$, perché $F(n+1) = F(n) + F(n-1)$ vale solo per $n \geq 1$. Allora in questo caso devo usare una base più ampia: $n_0 = 0, k = 1$.

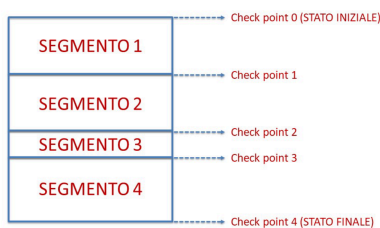
2.5.2. Correttezza

Per provare la correttezza di terminazione, cioè appunto la terminazione, è necessario assicurarsi che i cicli (inclusi i GOTO) e l'eventuale ricorsione abbiano termine.

Gli step dell'approccio generale alla soluzione del problema computazionale sono:

- *Definire lo stato iniziale* dell'algoritmo e quello *finale* che esso *deve raggiungere*;
- *Decomporre l'algoritmo in segmenti* e definire per ogni segmento lo stato in cui l'algoritmo si deve trovare del termine del segmento (*checkpoint*);
- *Dimostrare* che a partire dallo stato iniziale *si raggiungono* in successione *gli stati specificati* per la fine di ogni segmento. In particolare, lo stato che deve valere alla fine dell'ultimo segmento deve coincidere (o implicare) lo stato finale desiderato.

I *segmenti notevoli* sono cicli (for, while, repeat-until).



! Osservazione

I cicli sono i segmenti più difficili da analizzare (perché definiscono *implicitamente* molte operazioni) ma, insieme alla ricorsione, costituiscono gli strumenti essenziali per la scrittura di algoritmi o programmi interessanti.

2.5.3. Invariante

Per provare la correttezza di un ciclo (for, while, repeat-until, ...) bisogna dimostrare che al termine della sua esecuzione vale una certa *proprietà \mathcal{L}* che rappresenta uno stato ed è *funzionale alla correttezza dell'algoritmo*. A tal fine si fa uso di un *invariante*.

Un *invariante* per un ciclo è una *proprietà* espressa in funzione delle variabili usate nel ciclo, che descrive lo *stato* in cui si trova l'esecuzione alla fine di una generica iterazione del ciclo.

📌 Nota bene

L'invariante deve essere scelto finalizzandolo alla correttezza e alla prova della proprietà \mathcal{L} .

2.5.4. Correttezza di un ciclo tramite invariante

Per dimostrare che alla fine del ciclo vale una certa proprietà \mathcal{L} , si individua un opportuno invariante e si dimostra che:

1. Esso *vale all'inizio del ciclo* (subito prima che il ciclo inizi);
2. Esso *vale alla fine di ciascuna iterazione*, che si dimostra induttivamente provando che se vale alla fine di una generica iterazione, vale alla fine della successiva;
3. *Alla fine dell'ultima iterazione, l'invariante implica la proprietà \mathcal{L}* (in alcuni casi coincide con essa).

Terminologia

L'esecuzione di un *ciclo* consiste di più di zero iterazioni delle istruzione in esso contenute (*corpo del ciclo*).

Ad esempio `for i<-1 to n do {corpo del ciclo}` è un ciclo che esegue sempre n iterazioni.

2.5.4.1. Esempio (arrayMax)

Algoritmo `arrayMax(A)`

Input: Array $A[0 \div n-1]$ di $n \geq 1$ interi.

Output: massimo intero di A .

```
currMax<-A[0];
for i<-1 to n-1 do{
    currMax<-max{currMax, A[i]};
}
return currMax;
```

Proprietà \mathcal{L} : alla fine del ciclo, *currMax* è il massimo intero in A (quindi il valore restituito è corretto).

Invariante: Alla fine della iterazione $i \geq 0$,
 $currMax = \max\{A[0], A[1], \dots, A[i]\}$.

Osservazione

La fine dell'iterazione $i=0$ corrisponde all'inizio del ciclo.

1. All'inizio del ciclo ($i=0$) l'invariante è reso vero dall'assegnamento $currMax < -A[0]$;
2. Disponiamo l'invariante uguale a vero a fine iterazione $i < n-1 \implies currMax = \max\{A[0], \dots, A[i]\}$ e dimostriamo che vale anche alla fine della successiva iterazione.
Nella iterazione $i+1$ l'istruzione $currMax <- \max\{currMax, A[i+1]\}$ assicura che alla fine di tale iterazione $currMax <- \max\{A[0], \dots, A[i+1]\}$;
3. Alla fine dell'ultima istruzione ($i=n-1$):
Se vale l'invariante, cioè se $currMax = \max\{A[0], \dots, A[n-1]\}$, allora $currMax$ è effettivamente il massimo intero, che è la proprietà \mathcal{L} .

2.5.4.2. Esempio (arrayFind(A))

Algoritmo arrayFind(A)

Input: Elemento x , array $A[0 \div n-1]$ di n elementi.

Output: indice $i \in [0, n)$ tale che $A[i] = x$, se esiste, altrimenti -1 .

```
i <- 0;
while i < n do{
    if(x = A[i]) then return i;
    else i <- i+1;
}
return -1;
```

Proprietà \mathcal{L} : Il valore trovato è corretto.

Invariante: Alla fine di una generica iterazione $x \neq A[j] \ \forall 0 \leq j < i$ con i valore della variabile omonima alla fine dell'iterazione corrente.

1. All'inizio del ciclo ($i=0$) l'invariante vale per *vacuità* dato che il range $0 \div i-1$ è vuoto;
2. Supponiamo l'invariante vero alla fine di una generica iterazione $\implies x \neq A[j] \ \forall 0 \leq j < i < n$.

Alla fine della successiva iterazione:

- Se $x = A[i] \implies i$ non cambia e si esce dal ciclo;
- Se $x \neq A[i] \implies i$ diventa $i+1$.

In entrambi i casi l'invariante continua a valere;

3. Fine ultima iterazione, ci sono due possibili uscite:

- $u_1: x = A[i]$: In questo caso si restituisce i e chiaramente \mathcal{L}

vale;

- $u_2:i=n$ in questo caso si restituisce -1 .

Dato che l'invariante mi dice che $x \neq A[j] \quad \forall j: 0 \leq j < n=i$ so che x non è in A e quindi restituire -1 è corretto $\implies \mathcal{L}$ vale.

2.5.6. Ricorsione

Un *algoritmo ricorsivo* è un algoritmo che invoca sé stesso (su istanze sempre più piccole) sfruttando la nozione di induzione.

La soluzione di un'istanza di taglia n è ottenuta *direttamente* se $n = n_0, n_0 + 1, \dots, n_0 + k$ (casi base), altrimenti *riducendosi* alla soluzione di $r \geq 1$ istanze di taglia minore di n : se $n > n_0 + k$. Se $r = 1$ si parla di *linear recursion*.

2.5.6.1. Esempi

Algoritmo `linearSum(A, n)`

Input: array A , intero $n \geq 1$.

Output: $\sum_{i=0}^{n-1} A[i]$.

```
if (n = 1) then{
    return A[0];
}
else{
    return linearSum(A, n-1) + A[n-1];
}
```

Taglia dell'istanza: n ; $n_0 = 1$ e $k = 0$.

Se voglio trovare la somma di tutti gli elementi di un array A , la prima invocazione sarà `linearSum(A, |A|)`.

Algoritmo `reverseArray(A, i, j)`

Input: array A , indici $i, j \geq 0$.

Output: array A con gli elementi in $A[i \div j]$ ribaltati.

```
if(i < j) then{
    swap(A[i], A[j]);
    reverseArray(A, i+1, j-1);
}
return
```

Taglia dell'istanza: $n = j - i + 1$.

Caso base: $n \leq 1$.

La prima invocazione per ribaltare tutto A è `reverseArray(A,0,|A|-1)`.

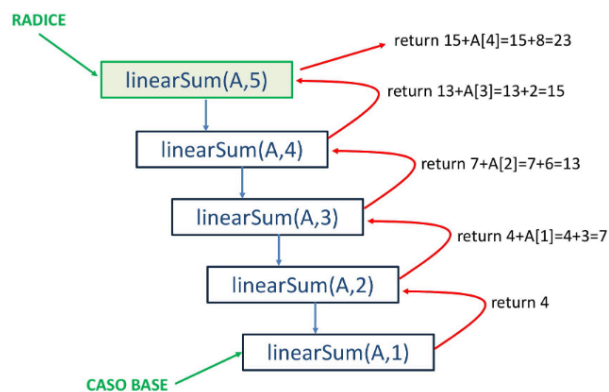
Lezione 06

2.5.6.2. Esecuzione di un algoritmo ricorsivo

All'esecuzione di un algoritmo ricorsivo su una data istanza è associato un *albero della ricorsione* (o *recursion trace*) tale che:

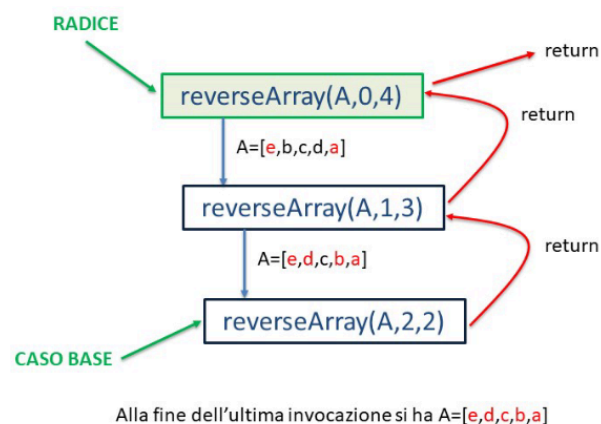
- Ogni nodo corrispondente a un'invocazione ricorsiva distinta fatta durante l'esecuzione dell'algoritmo;
- La *radice* dell'albero corrisponde alla prima invocazione, i *figli* di un nodo x sono associati alle invocazioni ricorsive fatte direttamente dall'invocazione corrispondente a x ;
- Le *foglie* dell'albero rappresentano i *casi base*.

L'albero della ricorsione per `linearSum(A,5)`, con $A = [4,3,6,2,8]$ risulta:



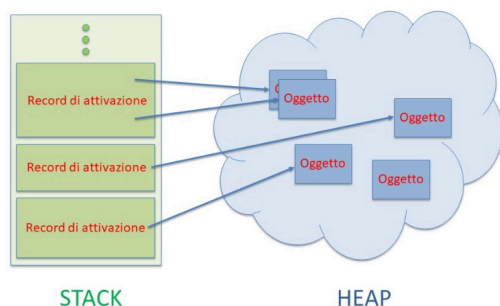
Avvengono un numero costante di operazioni per chiamata, quindi il numero totale sarà proporzionale al numero di chiamate. La complessità è allora $\Theta(n)$ (giustificazione a seguire).

L'albero della ricorsione per `reverseArray(A,0,4)`, con $A = [a,b,c,d,e]$ sarà invece:

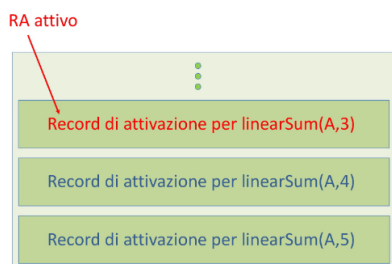


Nell'esecuzione di un programma (in Java) entrano di solito in gioco due spazi di memoria:

- **Stack**: spazio destinato a memorizzare variabili locali ai metodi e riferimenti a oggetti.
 - Per ogni invocazione di un metodo viene inserito un **record di attivazione** (abbreviato in RA; in inglese si usa il termine **activation record** o **activation frame**) contenente variabili e riferimenti a oggetti relativi a quell'invocazione.
 - Un RA viene eliminato quando l'invocazione del metodo corrispondente finisce l'esecuzione.
 - I RA sono inseriti/eliminati con politica LIFO (Last In First Out); in ogni istante possono essere acceduti i dati relativi all'ultimo RA inserito, ma non quelli di altri RA.
- **Heap**: spazio destinato a memorizzare gli oggetti.



Supponiamo di eseguire `linearSum(A,5)` e consideriamo l'invocazione ricorsiva `linearSu, (A,3)`. Quando viene creato il RA per tali invocazione ricorsiva, lo stato della Stack è il seguente:



Ricorda

Un algoritmo ricorsivo è un algoritmo che invoca sé stesso su istanze più piccole.

Un **algoritmo iterativo** è un algoritmo non-ricorsivo.

Osservazione

Il termine **iterativo** è usato per evidenziare che (tranne per casi banali) **un algoritmo non-ricorsivo fa uso di cicli per poter elaborare istanze di qualsiasi taglia**.

Mentre i cicli sono essenziali in un algoritmo iterativo, un

algoritmo ricorsivo può non avere cicli (come `linearSum` e `reverseArray`), ma può anche averli (come `MergeSort`).

2.5.7. Complessità di algoritmi ricorsivi

La *complessità* di un algoritmo ricorsivo A può essere *stimata tramite l'Albero della Ricorsione*.

Consideriamo l'albero associato all'esecuzione di A su un'istanza i di taglia n :

- A ogni nodo è attribuito un *costo* pari al numero di operazioni eseguite dall'invocazione corrispondente a quel nodo, *escluse quelle fatte dalle invocazioni ricorsive al suo interno*;
- Il numero totale di operazioni eseguite da A per risolvere i si ottiene sommando i costi associati a tutti i nodi.

Per ottenere un *upper bound* a $t_A(n)$ si ricava una stima per eccesso del numero totale di operazioni che valga per tutte le istanze i di taglia n , mentre per ottenere un *lower bound* a $t_A(n)$ si trova un'istanza particolare o si fa una stima inferiore che vada bene per tutte le istanze.

2.5.7.1. Esempi

2.5.7.1.1. Complessità di `linearSum(A,n)`

Algoritmo `linearSum(A,n)`

Input: array A , intero $n \geq 1$.

Output: $\sum_{i=0}^{n-1} A[i]$.

```
if (n = 1) then{
    return A[0];
}
else{
    return linearSum(A, n-1) + A[n-1];
}
```

L'albero della ricorsione associato a una generica istanza di taglia n :

- n nodi associati a `linearSum(A,j)`, $j = n-1, \dots, 1$;
- Costo associato a ciascun nodo: $\Theta(1)$ operazioni (esclude quello delle chiamate ricorsive)–

\Rightarrow Ogni istanza di taglia n richiede $\Theta(n)$ operazioni.

$\Rightarrow t_{\text{linearSum}}(n) \in \Theta(n)$ è la complessità al caso pessimo.

2.5.7.1.2. Complessità di `reverseArray(A, i, j)`

Algoritmo `reverseArray(A, i, j)`

Input: array A , indici $i, j \geq 0$.

Output: array A con gli elementi in $A[i \div j]$ ribaltati.

```
if(i < j) then{
    swap(A[i], A[j]);
    reverseArray(A, i+1, j-1);
}
return
```

L'albero della ricorsione associato a una generica istanza di taglia n ($n = j - i + 1$):

- $\lfloor \frac{n}{2} \rfloor + 1$ nodi associati;
- Costo associato a ciascun nodo: $\Theta(1)$ operazioni (esclude quello delle chiamate ricorsive).

\Rightarrow Ogni istanza di taglia n richiede $\Theta(n)$ operazioni.

$\Rightarrow t_{\text{reverseArray}}(n) \in \Theta(n)$ è la complessità al caso pessimo.

2.5.7.2. Calcolo efficiente di potenze

Dato $x \in \mathbb{R}$ e $n \geq 0$ intero, calcolare $p(x, n) = x^n$.

È fondamentale osservare che

$$p(x, n) = \begin{cases} 1 & n = 0 \\ x \cdot p(x, \frac{n-1}{2})^2 & n > 0 \text{ dispari} \\ p(x, \frac{n}{2})^2 & n > 0 \text{ pari} \end{cases}$$

Algoritmo `power(x, n)`

Input: $x \in \mathbb{R}$ e $n \geq 0$.

Output: $p(x, n)$.

```
if(n = 0) then{
    return 1;
}
if(n è dispari) then{
    y <- power(x, (n-1)/2);
    return x*y*y;
}
else{
```

```

    y <- power(x, n/2);
    return y*y;
}

```

2.5.7.2.1. Complessità di `power(x,n)`

Supponiamo $n \geq 1$ (consideriamo n come taglia dell'istanza):

- Alla i -esima chiamata ricorsiva l'algoritmo viene invocato per un esponente $n_i \leq \frac{n}{2^i}$. Di conseguenza l'albero della ricorsione avrà $O(\log n)$ nodi;
- Ogni invocazione di `power` esegue $\Theta(1)$ operazioni, esclusa l'eventuale chiamata ricorsiva. Quindi il costo associato a ciascun nodo è $\Theta(1)$.
 $\implies t_{\text{power}}(n) \in O(\log n)$.

2.5.7.2.2. Applicazione di `power` al calcolo di $F(n)$

Il seguente algoritmo efficiente sfrutta la formula $F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ e l'algoritmo `power` visto in precedenza:

Algoritmo `powerFib(n)`

Input: intero $n \geq 0$.

Output: $F(n)$.

```

ψ <- ((1 + sqrt{5})/2);
ψ^ <- ((1 - sqrt{5})/2);
return (power(ψ, n) - power(ψ^, n))/sqrt{5};

```

Da quanto provato per `power`, otteniamo che la complessità di `powerFib` è $O(\log n)$.

2.5.8. Correttezza di algoritmi ricorsivi

Per provare la correttezza (o una qualsiasi proprietà) di un algoritmo ricorsivo A si ricorre all'induzione.

Sia n la taglia dell'istanza:

1. Si dimostra la correttezza per i casi base $n \in [n_0, n_0 + k]$;
2. Supponendo che A risolva correttamente tutte le istanze di taglia $m \in [n_0, n]$, per un qualche $n \geq n_0 + k$, si dimostra che esso risolve correttamente tutte le istanze di taglia $n+1$.

2.5.8.1. Correttezza di `linearSum(A,n)` per $n \geq 1$

Caso base ($n=1$): correttezza banale.

Passo induttivo: Fisso $n \geq 1$ arbitrario.

Ipotesi induttiva: `linearSum(A,j)` sia corretto $\forall A, \forall 1 \leq j \leq n$.

Considero `linearSum(A,n+1)` con A array di $\geq n+1$ elementi:

`linearSum(A,n+1)` restituisce $\text{linearSum}(A,n) + A[n] = \sum_{i=0}^{n-1} A[i] + A[n] = \sum_{i=0}^n A[i]$
per ipotesi induttiva.

\Rightarrow Il valore restituito è corretto.

Riepilogo sulle nozioni di base

- Nozioni di: problema computazionale, algoritmo (che risolve un problema computazionale), taglia dell'istanza, struttura dati;
- Specifica di un algoritmo tramite pseudocodice;
- Complessità al caso peggio di un algoritmo:
 - Definizione;
 - Analisi asintotica espressa tramite ordini di grandezza ($O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$);
- Tecniche di dimostrazione: esempio, controesempio, per assurdo, induzione;
- Invarianti e loro uso per provare la correttezza di cicli;
- Algoritmi ricorsivi e loro analisi:
 - Complessità: tramite l'albero della ricorsione o tramite guess e induzione;
 - Correttezza: tramite induzione.

[lezione 07 & 08] - svolgimento esercizi

[Lezione 09](#)

3. Ripasso di Java

Per scrivere un programma stand-alone, chiamato `MyProgram.java` è necessario che la `public class` abbia lo stesso nome del file e che abbia il metodo `main`, dal quale inizia l'esecuzione.


```
import ... ; // import a package or a class
public class MyProgram {
    public static void main(String[] args) {
        /*...*/ // corpo del metodo main
    }
    /* eventuali altri metodi */
}
```

Per compilare il programma si usa il comando `javac MyProgram.java` da terminale, che crea un `bytecode` `MyProgram.class` eseguibile sulla *Java Virtual Machine* (JVM). Per eseguirlo si usa il comando `java MyProgram`, che richiede che la directory corrente sia nel `CLASSPATH`, altrimenti si usa `java -cp - MyProgram`. In alternativa, si può usare in *Integrated Development Environment* (IDE), come IntelliJ IDEA, Eclipse, Jbuilder.

3.1. Caratteristiche di Java e dell'approccio Object-Oriented

Le caratteristiche principali sono la modularità, l'astrazione e incapsulamento (information hiding) e l'ereditarietà (Inheritance).

3.1.1. Modularità

Le classi vengono viste come un insieme di oggetti che interagiscono insieme; Main è la classe principale, mentre le altre rappresentano tipi di oggetti. Un'*applicazione* è un insieme di *oggetti interagenti*.

Un *oggetto* è un'*istanza di una classe* che definisce variabili di istanza (in inglese "instance variables" o "fields") e metodi ("methods").

Per definire una variabile `i` di tipo `Integer`, alla quale assegno un oggetto di tipo `Integer`, dopo averlo creato, scrivo: `Integer i = new Integer(5);`.

3.1.2. Astrazione e incapsulamento (information hiding)

Con l'*information hiding* si astrae la *specifica* delle funzionalità, separandola dalla loro *implementazione*, quindi si possono usare le funzionalità solamente in base alla specifica.

Visto che l'implementazione delle funzionalità è nascosta, gli errori sono più confinati (si ha *robustezza*) e c'è la possibilità di cambiare l'implementazione senza cambiare la specifica (si ha

adattabilità).

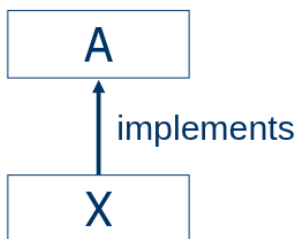
Nel caso delle strutture dati, questo approccio dà origine alla nozione di *Abstract Data Type* (ADT), che in Java si realizza tramite l'utilizzo di interfacce, classi e classi astratte.

Un'*interfaccia* è un insieme di dichiarazioni di metodi senza corpo e di costanti.

Una *classe* è la definizione di costanti, variabili e metodi con corpo.

Una *classe astratta* è una via di mezzo tra un'interfaccia e una classe (ci sono alcuni metodi senza corpo).

```
public interface A {  
    public int a1();  
    public boolean a2();  
}  
  
public class X implements A {  
    public int a1() { /*...*/ }  
    public boolean a2() { /*...*/ }  
    public void b() { /*...*/ }  
}  
  
A var1 = new A(); //NO!  
A var1 = new X(); //OK!
```



Ad esempio: `java.lang.String implements java.lang.Comparable`.

⚠ Attenzione

Non si può creare un oggetto di un'interfaccia, sono legati solo alle classi.

3.1.3. Ereditarietà (inheritance)

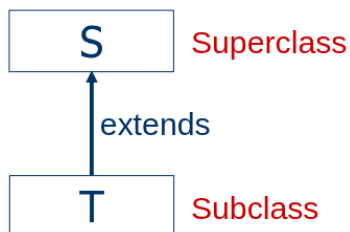
Con l'ereditarietà si *importano* in una classe le caratteristiche di un'altra classe, aggiungendone di nuove e/o specializzandone alcune. In Java una *sottoclasse estende una superclasse*

```

public class S {
    public void a() { /*...*/ }
    public void b() { /*...*/ }
    public int c() { /*...*/ }
}

public class T extends S {
    /* inherits b() and c() */
    float y; // added
    public void a() { /*...*/ } // specialized
    public boolean d() { /*...*/ } // added
}

```



! Osservazione

Ogni classe estende implicitamente `java.lang.Object`.

3.1.3.1. Ereditarietà multipla per interfacce

Un'Interfaccia (*non* una classe) può estender più interfacce.

```

public interface A { /*...*/ }
public interface B { /*...*/ }
public interface C extends A, B { /*...*/ }

```

! Osservazione

`A` e `B` non possono contenere metodi con la stessa firma.

```

public class D implements A, B {
    // deve implementare tutti i metodi di A e B
}

public class D implements C {
    // deve implementare tutti i metodi di A e B
}

```

```
// e quelli (eventuali) aggiuntivi di C
}
```

3.2. Programmazione generica (generics)

La programmazione generica è un tipo di polimorfismo, introdotto da Java SE5. Permette l'uso di *variabili di tipo* nella definizione di classi, interfacce e metodi. Si parla in questo caso di *classi/interfacce generiche* e *metodi generici*.

Nel creare un'istanza di una classe generica si specifica il tipo (*actual type*) da sostituire con la variabile di tipo. L'actual tupe non può essere un tipo primitivo (come per esempio un intero), ma deve essere una classe che estende `Object`.

Nell'invocazione di un metodo generico, la sostituzione della variabile di tipo con un actual type è fatta automaticamente in base ai parametri passati.

L'uso delle classi/interfacce generiche evita il ricorso a parametri di tipo `Object` e l'uso frequente di cast.

```
public interface MyInterface<E> {
    public int size();
    public boolean method1 (E var1);
    public E method2 ();
}

public class MyClass<E> implements MyInterface<E>{
    E var;
    public int size() { /*...*/ };
    public boolean method1 (E var1) { /*...*/ };
    public E method2 () { /*...*/ };
    public E method3 (float var2) { /*...*/ };
}

MyInterface<Integer> x = new MyClass<Integer>();

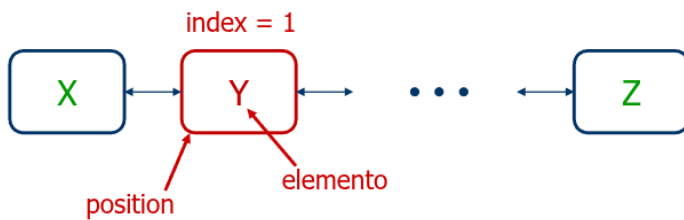
public class MyClass1<E extends S> { /*...*/ } // E può essere sostituito
solo da oggetti di tipo che estende/implementa la classe/interfaccia S
```

3.3. ADT elementari

3.3.1. Lista (list)

Una *lista* (*list*) è una collezione di elementi organizzati secondo un ordine lineare tale per cui è identificabile il primo elemento, il secondo e così via fino all'ultimo.

- In una lista *index-based* un elemento può essere acceduto tramite un *indice intero* che indica il numero di elementi che lo precedono;
- In una lista *position-bases* ogni elemento è contenuto in un contenitore detto *nodo* (*position*).



3.3.1.1. Lista index-based

```
public interface List<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Inserts an element e to be at index i, shifting all elements
    after this. */
    public void add(int i, E e);
    /** Returns the element at index i, without removing it. */
    public E get(int i);
    /** Replaces the element at index i with e, returning the previous
    element at i. */
    public E set(int i, E e);
    /** Removes and returns the element at index i shifting left
    subsequent elements. */
    public E remove(int i)
}
```

❗ Osservazione

I metodi `public int size()` e `public boolean isEmpty()` si troveranno in ogni ADT che vedremo.

Si implementa questo tipo di lista *tramite array*. I metodi possono essere implementati con complessità $O(1)$, tranne `add` e `remove`, che richiedono complessità $O(n)$ (dove n è il numero di elementi nella lista).

❗ Osservazione

Nel caso in cui l'array sia pieno, l'inserimento di un elemento x è implementato creando un nuovo array di capacità maggiore (solitamente doppia), trasferendo tutte le entry dal vecchio al nuovo array, e inserendo l'elemento x nel nuovo array.

3.3.1.2. Lista position-based

```
public interface Position<E> {
    /** Return the element stored at this position */
    public E getElement();
}

public interface PositionalList<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Returns the first node in the list. */
    public Position<E> first();
    /** Returns the last node in the list. */
    public Position<E> last();
    /** Returns the node after a given node in the list. */
    public Position<E> after(Position<E> p);
    /** Returns the node before a given node in the list. */
    public Position<E> before(Position<E> p);
    /** Inserts an element at the front of the list. */
    public void addFirst(E e);
    /** Inserts an element at the back of the list. */
    public void addLast(E e);
    /** Inserts an element after the given node in the list. */
    public void addAfter(Position<E> p, E e);
    /** Inserts an element before the given node in the list. */
    public void addBefore(Position<E> p, E e);
    /** Removes a node from the list, returning the element stored there. */
    public E remove(Position<E> p);
    /** Replaces the element stored at the given node, returning old element. */
    public E set(Position<E> p, E e);
}
```

Si implementa questo tipo di lista tramite *doubly-linked* list. Ogni nodo diventa un oggetto a sé, con variabili di istanza che "puntano" al predecessore e al successore. L'inizio e la fine della lista sono delimitati da dei *nodi sentinella*. I metodi possono essere implementati con complessità $O(1)$, ma la lista può essere scandita solo sequenzialmente,

❗ Osservazione

È possibile implementare una lista index-based tramite *souble-linked* list, o una lista position-based tramite array, ma con scarsi vantaggi.

3.3.2. Pila (stack) e coda (queue)

Una *pila* (*stack*) è una collezione di elementi inseriti e rimossi in base al principio *Last-In First-Out* (LIFO).

```
public interface Stack<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Inserts an element e at the top of the stack. */
    public void push(E e);
    /** Returns the element at the top of the stack without removing it.
    */
    public E top();
    /** Removes and returns the element at the top of the stack. */
    public E pop();
}
```

Una *coda* (*queue*) è una collezione di elementi inseriti e rimossi in base al principio *First-In First-Out* (FIFO).

```
public interface Queue<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Inserts an element e at the rear of the queue. */
    public void enqueue(E e);
}
```

```

    /** Returns but does not remove the first element of the queue (null
    if empty). */
    public E first();

    /** Removes and returns the first element of the queue (null if
    empty). */
    public E dequeue();
}

```

Entrambe le strutture dati possono essere implementate tramite *doubly-linked list*. La pila effettua inserimento (*push*) e rimozione (*pop*) in testa alla lista, mentre la coda effettua l'inserimento (*enqueue*) in coda e la rimozione (*dequeue*) in testa alla lista. Tutti i metodi possono essere implementati con complessità $O(1)$ ma, a differenza della lista, non si ha accesso a element intermedi a meno di no rimuovere quelli che li precedono nell'ordine di accesso.

❗ Osservazione

È possibile usare anche una *singly-linked list*.

3.4. Collection framework di Java

"*Collection*" è un termine generico per indicare una struttura dati.

Il *collection framework di Java* è un'architettura unificata di strutture dati e algoritmi implementato nel package `java.util`.

Esso contiene:

- *interfacce* di varie collection;
- *classi* che implementano le interfacce;
- *algoritmi polimorfi* per operare su collection (*metodi statici della classe Collections*), come ad esempio il sorting;
- ampio uso della *programmazione generica*.

Il package `java.util` contiene diverse implementazioni di Liste, Pile e Code. Tra esse si segnalano le seguenti:

- Lista
 - Interfaccia: `List`;
 - Classe `ArrayList`: implementazione index-based;
 - Classe `LinkedList`: implementazione sia index-based che position-based. Tuttavia l'implementazione position-based non

utilizza esplicitamente il concetto di position ma si basa sull'uso di iteratori.

- Pila e Coda
 - Interfaccia unificata: `Deque` (Double-ended queue);
 - Classe `LinkedList`: implementazione unificata di Pila e Coda (e Lista);
 - Classe `Stack`: implementazione della Pila basata su `Vector`. Si consiglia tuttavia l'uso di classi, come `LinkedList`, che implementano l'interfaccia `Deque` che è più completa.

Info

I nomi di alcuni metodi differiscono da quelli riportati precedentemente che seguono il libro di testo.

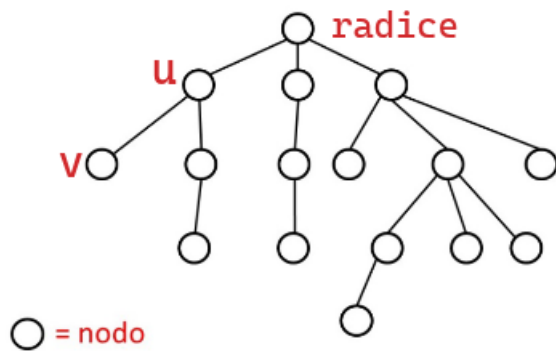
Riepilogo sul ripasso di Java

- Programma stand-alone;
- Caratteristiche di Java e dell'approccio Object-Oriented;
- ADT elementari:
 - Lista (index-based e position-based);
 - Pila
 - Coda
- Collection framework di Java, con particolare attenzione alle interfacce e classi che implementano Lista, Pila e Coda.

4. Alberi

4.1. Alberi generali

Un *albero* è una collezione di *nodi* caratterizzata da una *struttura gerarchica* che si dipana da un nodo *radice* tramite relazioni di tipo padre-figlio.



Nel disegno u è padre di v , v è figlio di u .

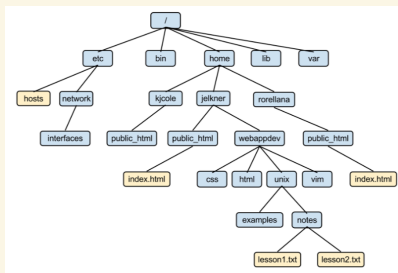
! Osservazioni

Le relazioni padre-figlio costituiscono un insieme di collegamenti minimali che introducono un legame (connessione) tra tutti i nodi.

Una lista è un caso estremo di albero con una struttura gerarchica lineare.

? Campi applicativi

1. Strutture dati: mappe, priority queue;
2. Esplorazione risorse: filesystem, siti di e-commerce;



3. Sistemi distribuiti e reti di comunicazione: sincronizzazione, broadcast, gathering;
4. Analisi di algoritmi: albero della ricorsione;
5. Classificazione: alberi di decisione;
6. Compressione di dati (codici di Huffman);
7. Biologia computazionale: alberi filogenetici.

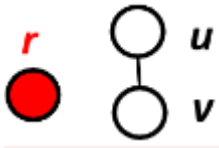
4.1.1. Definizioni e proposizioni

Un *albero radicato* (*rooted tree*) T è una collezione di nodi che, se non è vuota, soddisfa le seguenti proprietà:

- \exists un nodo speciale $r \in T$ (r è chiamato *radice*);

- $\forall v \in T, v \neq r: \exists! u \in T: u \text{ è padre di } v$ (v è figlio di u);
- $\forall v \in T, v \neq r$: risalendo di padre in padre si arriva a r (ovvero ogni nodo è discendente dalla radice).

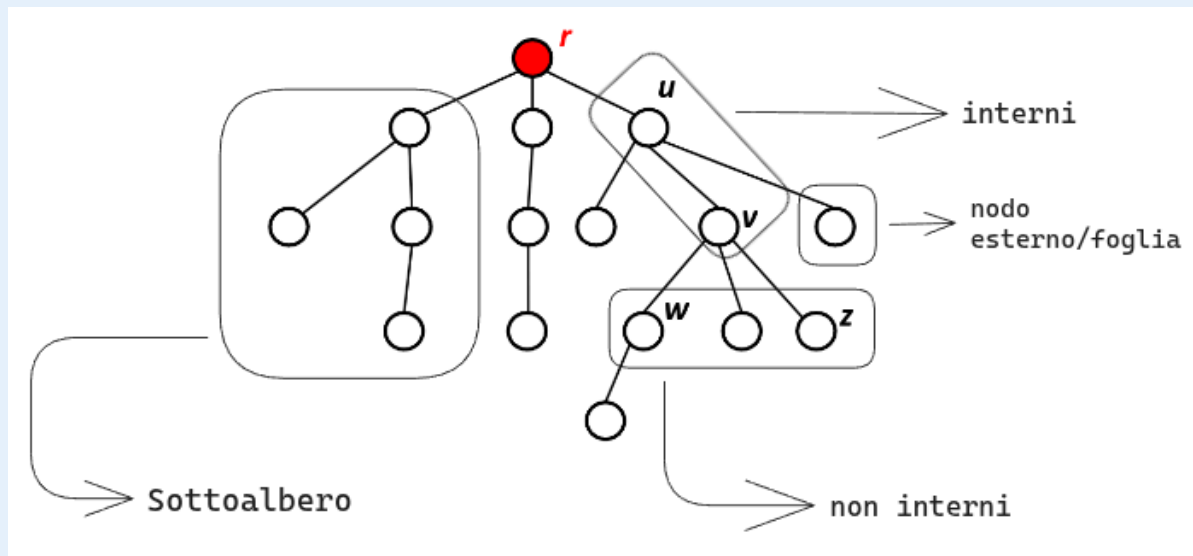
Nel libro di testo la terza condizione manca. Senza questa, la seguente collezione con u padre di v e v padre di u sarebbe un albero, che ha poco senso. Questa collezione piuttosto è una foresta di alberi.



Lezione 10

Terminologia

- x è *antenato* di y se $x=y$ oppure x è antenato del padre di y ;
- x è *discendente* di y se y è antenato di x ;
- I *nodi interni* sono quei nodi con almeno 1 figlio;
- I *nodi esterni* (o *foglie*) sono i nodi senza figli;
- T_v è un albero formato da tutti i discendenti di v (quindi include v);
- T è un *albero ordinato* se per ogni nodo interno $v \in T$ è definito un ordinamento lineare tra i figli u_1, u_2, \dots, u_k di v .



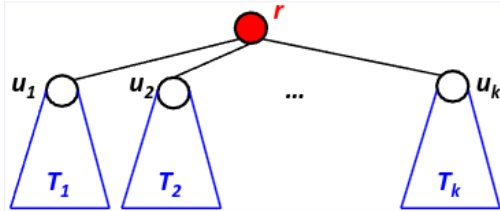
Si può definire ricorsivamente cos'è un albero radicato.

Un *albero radicato* T è una collezione di nodi che, se non è vuota, risulta partizionata in questo modo:

$$T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_k$$

per un qualche $k \geq 0$, dove:

- r è radice con figli u_1, u_2, \dots, u_k ;
- $\forall i, 1 \leq i \leq k$: T_i è un albero non vuoto con radice u_i ($\implies T_i \equiv T_{u_i}$).



Chiaramente, le due definizioni di albero radicato (ricorsiva e non) sono equivalenti.

Si definisce la *profondità* di un nodo v in un albero T in due modi alternativi:

1. $\text{depth}_T(v) = |\text{antenati}(v)| - 1$;
2.
 - Se $v = r$ radice $\implies \text{depth}_T(v) = 0$;
 - Altrimenti $\text{depth}_T(v) = 1 + \text{depth}_T(\text{padre}(v))$.

Il *livello* è l'insieme dei nodi a profondità i ($i \geq 0$).

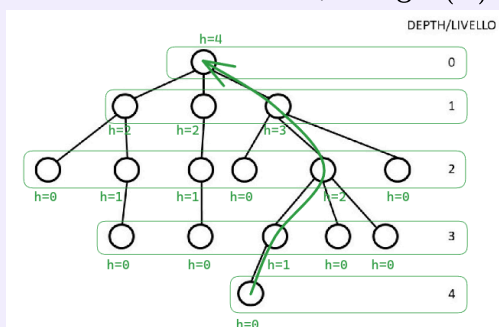
L'*altezza* di un nodo v in un albero T ($\text{height}_T(v)$) si definisce come:

- Se v è foglia $\implies \text{height}_T(v) = 0$;
- Altrimenti $\text{height}_T(v) = 1 + \max_{w: w \text{ figlio di } v} (\text{height}_T(w))$.

L'*altezza di un albero* T si definisce $\text{height}(T) = \text{height}_T(r)$, con r radice di T .

Proposizione

Dato un albero T , $\text{height}(T) = \max_{v \in T: v \text{ foglia}} (\text{depth}_T(v))$.

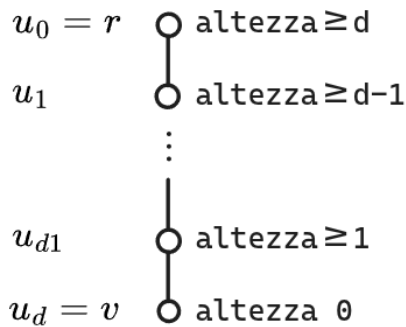


4.1.1.1. Dimostrazione

Sia h l'altezza dell'albero e $d = \max_{v \in T: v \text{ foglia}} (\text{depth}_T(v))$. Proviamo $h \geq d$ e $h \leq d$.

4.1.1.1.1. $h \geq d$

Sia v una foglia a profondità d e sia r la radice di T . allora esiste un percorso da v a r .

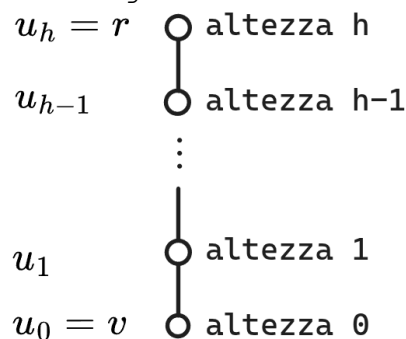


È immediato vedere che

$$\text{height}_T(u_i) \geq d - i \quad \forall d \geq i \geq 0 \implies \text{height}_T(r) \geq d \implies h \geq d.$$

4.1.1.1.2. $h \leq d$

Per assurdo, se $h \geq d$ deve esistere un percorso in T dalla radice r a una foglia v .



In questo caso, la foglia v ha h antenati diversi da v , che sono u_1, u_2, \dots, u_h , e quindi $\text{depth}_T(v) = h > d$, che contraddice il fatto che d è la massima profondità di una foglia.

□