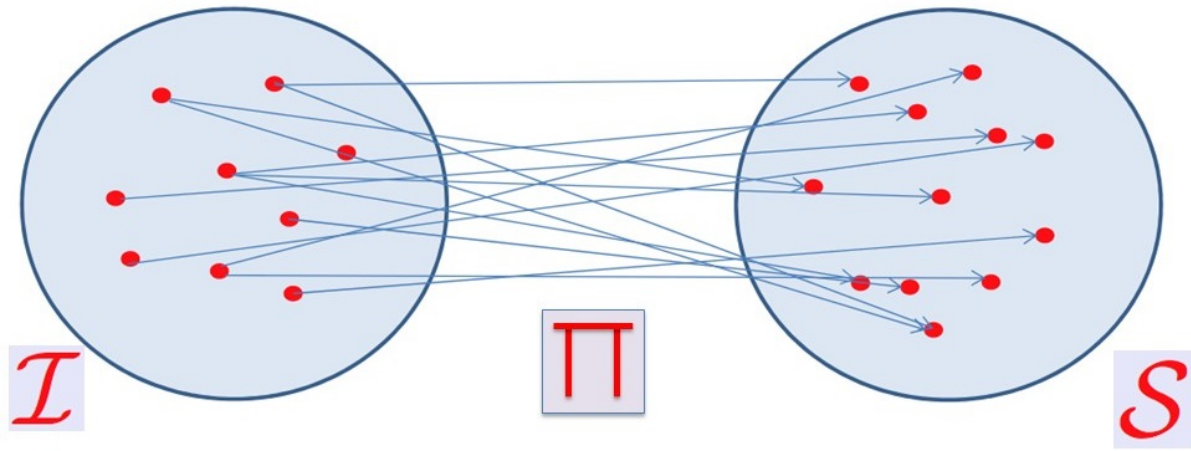
Dati e Algoritmi: A. Pietracaprina

Nozioni di Base

Argomenti trattati

- Nozioni di: problema computazionale, modello di calcolo, algoritmo, pseudocodice, struttura dati
- Analisi di complessità: analisi al caso pessimo, analisi asintotica, ordini di grandezza
- Analisi di correttezza: tecniche di dimostrazione, induzione, invarianti
- Algoritmi ricorsivi e loro analisi

Problema Computazionale



Un *problema computazionale* è costituito da

- un insieme $\mathcal{I}$ di ISTANZE (i *possibili input*)
- un insieme $\mathcal{S}$ di SOLUZIONI (i *possibili output*)
- una relazione Π che a ogni istanza $i \in \mathcal{I}$ associa *una o più* soluzioni $s \in \mathcal{S}$.

  Oss. Π è un *sottoinsieme del prodotto cartesiano* $\mathcal{I} \times \mathcal{S}$

Esempi

Somma di Interi (Z)

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

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.

$$(< 43, 16, 75, 2 >, < 2, 16, 43, 75 >) \quad \in \quad \Pi$$
$$(< 7, 1, 7, 3, 3, 5 >, < 1, 3, 3, 5, 7, 7 >) \quad \in \Pi$$
$$(< 13, 4, 25, 17 >, < 11, 27, 33, 68 >) \quad \notin \Pi$$

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.

$$(< 43, 16, 75, 2 >, < 3, 1, 0, 2 >) \in \Pi$$

$$(< 7, 1, 7, 3, 3, 5 >, < 1, 3, 4, 5, 0, 2 >) \in \Pi$$

$$(< 7, 1, 7, 3, 3, 5 >, < 1, 4, 3, 5, 0, 2 >) \in \Pi$$

$$(< 13, 4, 25, 17 >, < 0, 1, 3, 2 >) \notin \Pi$$

Osservazioni

- istanze diverse possono avere la stessa soluzione (ad es., somma)
- un'istanza può avere diverse soluzioni (ad es., ordinamento ver. 2)

Esercizio

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

Esercizio

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

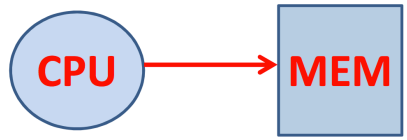
Algoritmo e modello di calcolo

Definizione

Algoritmo: procedura computazionale ben definita che transforma 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.

Modello di calcolo RAM [1] (*Random Access Machine*)



- input, output, dati intermedi (e programma): in memoria
- operazioni elementari: *assegnamento, operazioni logiche, operazioni aritmetiche, indicizzazione di array, return di un valore da parte di un metodo*, ecc.

[1] Diverso da *Random Access Memory*!

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

1. $\mathtt{A}$ calcola una funzione da $\mathcal{I}$ a $\mathcal{S}$, e quindi,
   - riceve come input istanze $i \in \mathcal{I}$
   - produce come output soluzioni $s \in \mathcal{S}$
2. Dato $i \in \mathcal{I}$, $\mathtt{A}$ produce in output $s \in \mathcal{S}$ tale che $(i, s) \in \Pi$.

Se Π associa più soluzioni a una istanza i, per tale istanza $\mathtt{A}$ ne calcola una (quale, dipende da come è stato progettato).

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.

USARE SEMPRE QUESTA STRUTTURA PER LO PSEUDOCODICE!!

Per maggiori dettagli fare riferimento alla dispesa sulla Specifica di Algoritmi in Pseudocodine, disponibile sul Moodle del corso.

Esempio: trasposta di una matrice $n \times n$ di interi

PROBLEMA COMPUTAZIONALE $\Pi \subseteq I \times S$

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

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

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

ALGORITMO: Idea per $n = 4$

Idea: scambiare ciascuna entry a_{ij} del triangolo superiore ($i = 0, 1, \ldots, n-2$ e $j > i$) con a_{ji}

Algoritmo Transpose(A)

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

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
```

Taglia di una istanza

L'analisi di un algoritmo (ad es., 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.

- **Ordinamento di un array:**

Taglia $n \equiv$ # elementi dell'array.

- **Trasposta di una matrice:** matrice $n \times m$

* Taglia $x = n \cdot m$
* Taglia $(n, m) \equiv$ # righe e # colonne
* (Se $n = m$) Taglia $n \equiv$ # righe/colonne

Strutture 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.

Definizione

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

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.

Ad esempio in Java:

- Livello logico: (gerarchia di) *interfacce*
- Livello fisico: (gerarchia di) *classi*.

Consegnare su Moodle Esami entro 8 Ottobre

Esercizio

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

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

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.

Analisi degli algoritmi

L'analisi di un algoritmo `A` mira a studiarne l'*efficienza* e l'*efficacia*. In particolare, essa può valutare i seguenti aspetti:

Complessità

- tempo
- spazio

Correttezza

- terminazione
- soluzione del problema computazionale

Noi ci concentreremo su:

1. complessità: tempo
2. correttezza: soluzione del problema computazionale

COMPLESSITÀ IN TEMPO

OBIETTIVO:

Stimare il tempo di esecuzione (*running time*) di un algoritmo, al fine di:

- valutarne l'efficienza
- poterlo confrontare con altri algoritmi per lo stesso problema.

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

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

* Richiede di implementare l'algoritmo con un programma e l'impatto della implementazione può influenzare il confronto tra algoritmi
* Non si possono considerare tutti gli input (soprattutto se infiniti)
* La stima dipende dall'ambiente HW/SW

APPROCCIO CHE SODDISFA I REQUISITI

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

Osservazione: esiste anche l'analisi al caso medio (*average case*), e l'analisi probabilistica

Sia $\mathtt{A}$ un algoritmo che risolve $\Pi \subseteq \mathcal{I} \times \mathcal{S}$.

Definizione

La complessità (in tempo) al caso pessimo di $\mathtt{A}$ è una funzione $t_{\mathtt{A}}(n)$ definita come *il massimo numero di operazioni elementari che $\mathtt{A}$ esegue per risolvere una istanza di taglia n*.

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

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

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.
- È difficile contare il max numero di operazioni richieste per risolvere tale istanza peggiore (il conteggio richiederebbe anche una specifica dettagliata del set di operazioni elementari del modello RAM).

È necessario determinare esattamente $t_A(n)$?

No, per le seguenti ragioni:

- Il tempo di esecuzione, che la complessità vuole stimare, dipende da tanti fattori che è impossibile quantificare in modo preciso.
- Le diverse operazioni elementari del modello RAM possono avere impatto diverso sui tempi di esecuzione a seconda delle architetture.

$\Rightarrow$ **ci accontentiamo di limiti superiori e limiti inferiori a $t_A(n)$.**

Esempio: `arrayMax`

Algoritmo `arrayMax(A)`
Input: array $A[0 \div n-1]$ di $n \geq 1$ interi
Output: max intero in A
currMax $\leftarrow A[0]$;
for $i \leftarrow 1$ *to* $n-1$ **do**
  if $(\mathrm{currMax} < A[i])$ **then** currMax $\leftarrow A[i]$;
return currMax

Taglia dell'istanza: n (ragionevole)

Stima di $t_{\texttt{arrayMax}}(n)$

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.

$\Rightarrow$ esistono costanti $c_1, c_2, c_3, c_4 > 0$ tali che per ogni n

$$\begin{aligned} t_{\texttt{arrayMax}}(n) &\leq c_1 n + c_2 \quad (\textit{limite superiore}) \\ t_{\texttt{arrayMax}}(n) &\geq c_3 n + c_4 \quad (\textit{limite inferiore}) \end{aligned}$$

Dobbiamo stimare le costanti c_1, c_2, c_3, c_4?

No, per le stesse ragioni per cui non è necessario stimare esattamente la complessità

Analisi Asintotica

Si ricorre quindi alla analisi asintotica, ignorando

- fattori moltiplicativi costanti (rispetto alla taglia dell'istanza)
- termini additivi non dominanti

Nel caso di `arrayMax` possiamo affermare che

- $t_{\texttt{arrayMax}}(n)$ è *al più* proporzionale a n (limite superiore)
- $t_{\texttt{arrayMax}}(n)$ è *almeno* proporzionale a n (limite inferiore)

Dalle due affermazioni consegue che:

$$t_{\texttt{arrayMax}}(n) \text{ è proporzionale a } n \text{ (limite stretto)}$$

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

Ordini di grandezza

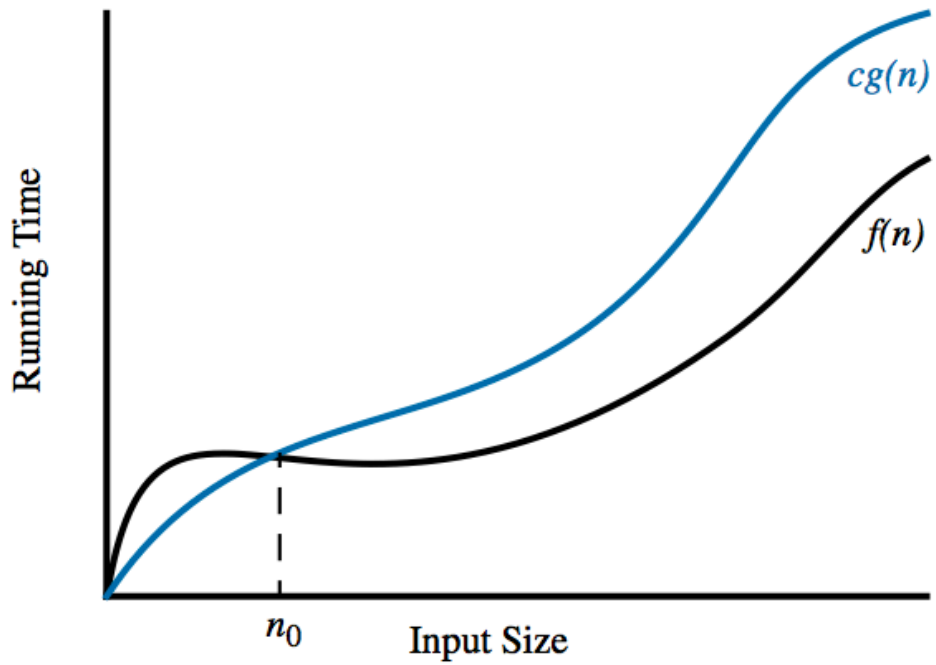
$f(n), g(n)$ funzioni da $\mathbb{N}$ a $\mathbb{R}^+ \cup \{0\}$.

Definizione

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

$$f(n) \leq cg(n), \forall n \geq n_0$$

A parole: $f(n)$ è AL PIÙ proporzionale a $g(n)$



Esempi

$f(n)$	$O(\cdot)$	c	n_0
$3n+4$ per $n \geq 1$	$O(n)$	4 (7)	4 (1)
$n+2n^2$ per $n \geq 1$	$O(n^2)$	3	1
2^{100} per $n \geq 1$	$O(1)$	2^{100}	1
$c_1 n + c_2$ per $n \geq 1$, $c_1, c_2 > 0$	$O(n)$	c_1+c_2	1

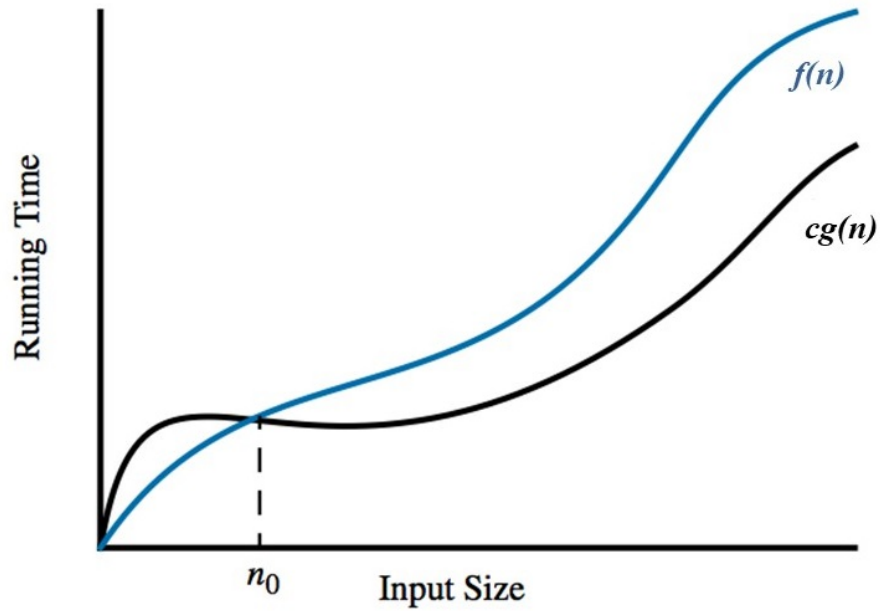
$3n+4 \in O(n^5)$ $\quad c=7 \quad n_0=1$

Definizione

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

$$f(n) \geq cg(n), \forall n \geq n_0$$

A parole: $f(n)$ è ALMENO proporzionale a $g(n)$



Esempi

$g(n)$

$f(n)$	$\Omega(\cdot)$	c	n_0
$3n+4$ per $n \geq 1$	$\Omega(n)$	1	1
$n+2n^2$ per $n \geq 1$	$\Omega(n^2)$	1	1
2^{100} per $n \geq 1$	$\Omega(1)$	2^{100}	1
$c_3 n + c_4$ per $n \geq 1$, $c_3, c_4 > 0$	$\Omega(n)$	1	1

$n+2n^2 \in \Omega(n)$? 1 1

Definizione

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

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

A parole: $f(n)$ è (ESATTAMENTE) proporzionale a $g(n)$

Applicando la definizione agli esempi delle slide precedenti otteniamo:

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

Definizione

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

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

In parole: $f(n)$ è ASINTOTICAMENTE più piccolo di $g(n)$

↘ ≡ cresce meno

Esempi

- $f(n) = 100n$ per $n \geq 1 \Rightarrow f(n) \in o\left(n^2\right)$
- $f(n) = 3n/(\log_2 n)$ per $n \geq 1 \Rightarrow f(n) \in o(n)$

Proprietà degli ordini di grandezza

1. $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$, per ogni $f(n), g(n) : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$.

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 es.: $(n+1)^5 \in \Theta(n^5)$

3. $\log_b n \in \Theta(\log_a n)$, se $a, b > 1$ sono costanti.

   Oss. 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 $h, k > 0$.

Strumenti matematici

- (**Parte bassa**) $\forall x \in \Re$, si definisce $\lfloor x \rfloor$ = più grande intero $\leq x$.

  Ad es.: $\lfloor 3/2 \rfloor = 1$; $\lfloor 3 \rfloor = 3$.

- (**Parte alta**) $\forall x \in \Re$, si definisce $\lceil x \rceil$ = più piccolo intero $\geq x$.

  Ad es.: $\lceil 3/2 \rceil = 2$; $\lceil 3 \rceil = 3$.

- (**Modulo**) $\forall x, y \in Z$, con $y \neq 0$, si definisce x mod y = resto della divisione intera x/y. (% in Java).

  Ad es.: $39 \bmod 7 = 4$; $80 \bmod 4 = 0$.

- (**Sommatorie notevoli**) $\forall n \in Z$ e $a \in \Re$, con $n \geq 0$ e $a > 0$ vale

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

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

$$\sum_{i=1}^{n} a^i = \frac{a^{n+1}-1}{a-1} - 1 = \frac{a^{n+1}-a}{a-1} \in \Theta(a^n) \quad \text{per } a>1$$

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 $t_A(n)$.

Limite Superiore (Upper Bound)

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

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

Limite Inferiore (Lower Bound)

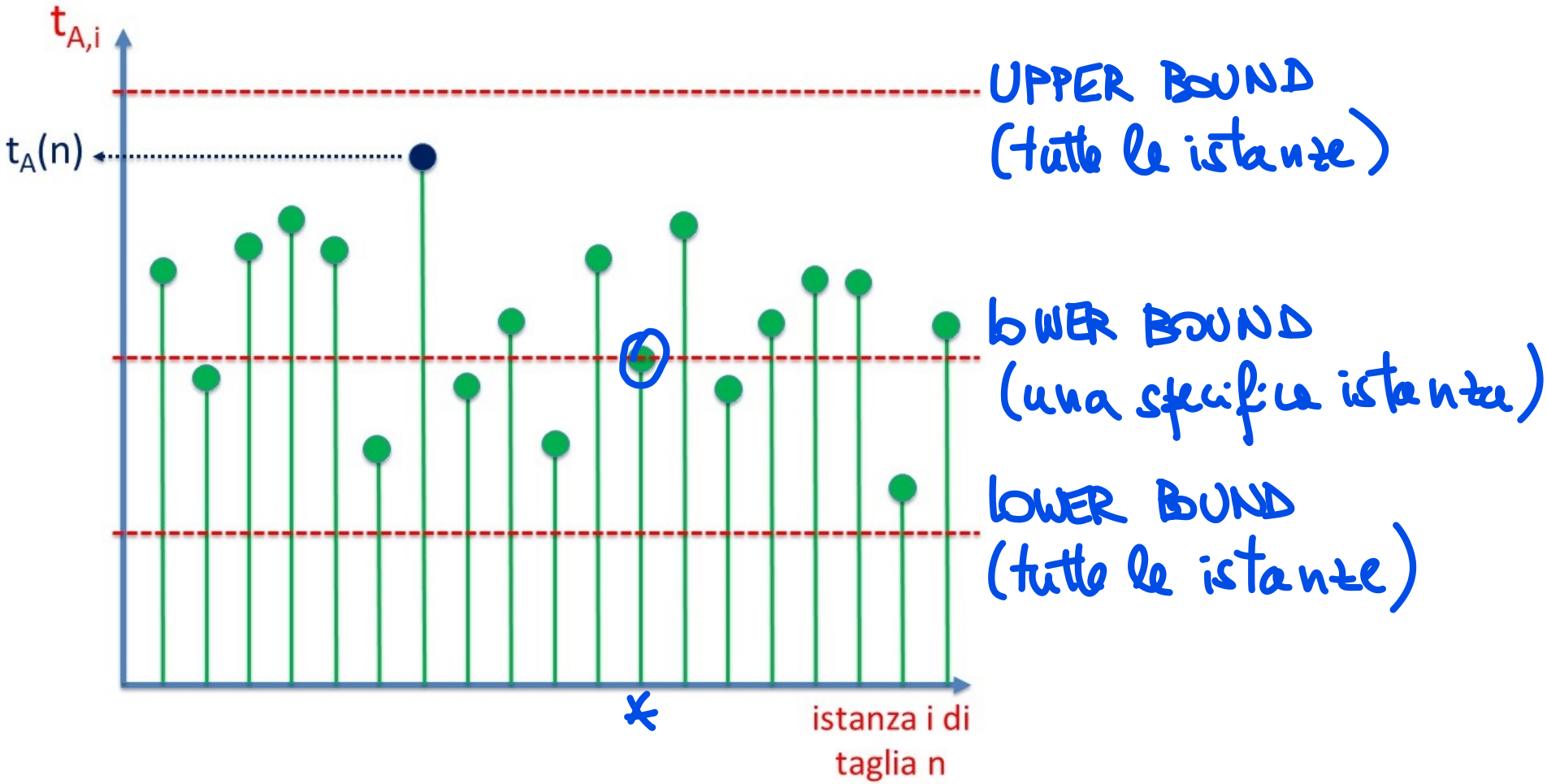
$$t_A(n) \in \Omega(f(n))$$

Si prova argomentando che per ogni n "abbastanza grande", *esiste* una istanza di taglia n per la quale l'algoritmo esegue $\geq cf(n)$ operazioni, con c costante (non serve determinarla)
In alcuni casi è comodo argomentare che per *ciascuna* istanza di taglia n l'algoritmo esegue $\geq cf(n)$ operazioni

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 $\Rightarrow$ no costanti, no termini additivi di ordine inferiore, ma *solo termini essenziali!*

Limiti superiori e inferiori



Terminologia per complessità [GTG14, Par. 4.2]

- 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

Esempio: Prefix Averages ([GTG14] pp. 164-165)

Si consideri il seguente problema computazionale: dato un array di n interi $X[0 \ldots n-1]$ calcolare un array $A[0 \ldots n-1]$ dove

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

Vedremo adesso 2 algoritmi: 1 banale e inefficiente, e 1 più furbo ed efficiente.

Per entrambi gli algoritmi vale la seguente specifica di input-output:

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

Output: $A[0 \ldots n-1]$: $A[i] = (\sum_{j=0}^{i} X[j])/(i+1)$ per $0 \leq i < n$

Algoritmo prefixAverages1

```
for i ← 0 to n − 1 do
    a ← 0;
    for j ← 0 to i do
        a ← a + X[j];
    A[i] ← a/(i+1);
return A
```

i+1 iterazioni con $\Theta(1)$ operazioni ciascuna

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

Complessità:

$$t_{PA1}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1}(i+1)\right) = \Theta\left(\sum_{i=0}^{n-1} i\right)$$

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

Algoritmo prefixAverages2

$s \leftarrow 0$;
for $i \leftarrow 0$ *to* $n-1$ **do**
    $s \leftarrow s + X[i]$;
    $A[i] \leftarrow \frac{s}{i+1}$;
return A

n iterazioni con $\Theta(1)$ operazioni ciascuna

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

Complessità

$$t_{PA2}(n) \in \Theta\left(1 + \sum_{i=0}^{n-1} 1\right) = \Theta(n)$$

Possiamo quindi affermare che $t_{PA2}(n) \in o(t_{PA1}(n))$ e quindi l'algoritmo prefixAverages2 è, al crescere di n, asintoticamente più efficiente di prefixAverages1

Esercizio

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 $\Theta(\cdot)$.

Taglia dell'istanza: n Complessità: $t_A(n)$

UPPER BOUND: # istruzioni di taglia n, A esegue

$\leq n \max\{c_1 n, c_2 \lceil \log_2 n \rceil\}$

$\leq n \max\{c_1, c_2\} \cdot n$ operazioni.

$\Rightarrow t_A(n) \in O(n^2)$ (1)

LOWER BOUND basato su una particolare istanza

Sia X un array di n interi pari

$\Rightarrow$ Per tale X, A esegue $n \cdot c_1 n = c_1 n^2$ operazioni

$\Rightarrow t_A(n) \in \Omega(n^2)$ (2)

(1) + (2) $\Rightarrow t_A(n) \in \Theta(n^2)$

LOWER BOUND basato su tutte le istanze

$\forall$ istanze di taglia n, A esegue

$\geq n \cdot \min\{c_1 n, c_2 \lceil \log_2 n \rceil\}$

$\geq n \cdot \min\{c_1, c_2\} \cdot \lceil \log_2 n \rceil$ operazioni

$\Rightarrow t_A(n) \in \Omega(n \log n)$

Esercizio

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 ... 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
}
```

Osservazione: 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.

Esercizio (continua)

1. Trovare delle opportune funzioni $f_1(n)$, $f_2(n)$ e $f_3(n)$ 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 $\leq cf_1(n)$ operazioni.
   - Per ciascuna istanza di taglia n l'algoritmo esegue $\geq cf_2(n)$ operazioni.
   - Esiste una istanza di taglia n per la quale l'algoritmo esegue $\geq cf_3(n)$ 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 $t_{\mathrm{IS}}(n)$ la complessità al caso pessimo dell'algoritmo. Sfruttando le affermazioni del punto precedente trovare un upper bound $O(\cdot)$ e un lower bound $\Omega(\cdot)$ per $t_{\mathrm{IS}}(n)$.

Regola di buon senso

Complessità polinomiale (o migliore) $\Rightarrow$ algoritmo efficiente

Complessità esponenziale $\Rightarrow$ algoritmo inefficiente

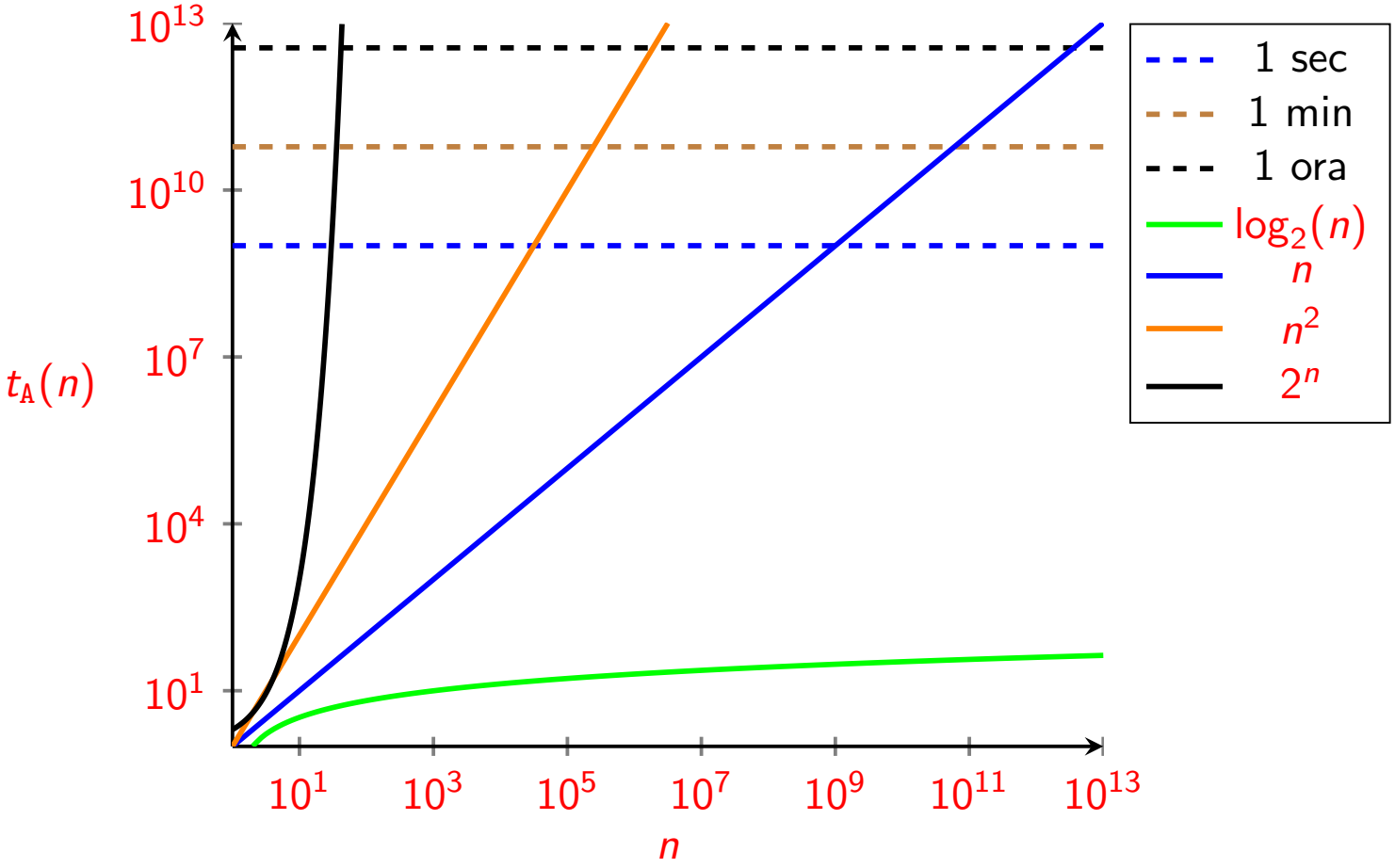
Giustificazione $\approx$[GTG14,4.3.2]

Supponiamo che la complessità $t_{\mathtt{A}}(n)$ sia espressa in ns, e definiamo

$$n_\tau \equiv \textit{max taglia di una istanza risolvibile in tempo } \tau.$$

Per ottenere n_τ, risolviamo $t_{\mathtt{A}}(n_\tau) = \tau$ rispetto a n_τ.

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


10^{13}
10^{10}
10^{7}
10^{4}
10^{1}
$t_A(n)$
10^1 10^3 10^5 10^7 10^9 10^{11} 10^{13}
n
1 sec
1 min
1 ora
$\log_2(n)$
n
n^2
2^n

Esempio

Crittografia a chiave pubblica: molto usata dai protocolli di sicurezza.

Invio di un messaggio cifrato da Alice a Bob:

- 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.

Integer factorization (per interi prodotti di 2 primi)

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

Quale taglia della istanza scegliamo?

Algoritmo banale

for $p \leftarrow 2$ *to* $\lfloor \sqrt{N} \rfloor$ **do**
  if $(N \bmod p = 0)$ **then return** $\{p, N/p\}$;

Complessità. La esprimiamo in funzione del numero n bit che servono per rappresentare N. Se $p, q \in \Theta\left(\sqrt{N}\right)$ la complessità è

$$\Theta\left(\sqrt{N}\right) = \Theta\left(2^{\frac{n}{2}}\right) \qquad \text{COMPLESSITÀ ESPONENZIALE !!}$$

Esempio numerico

- $n = 1024 \Rightarrow 2^{\frac{n}{2}} = 2^{512} \geq 10^{154}$
- Sul computer più potente al mondo ($< 10^{19}$ flop/sec) l'algoritmo banale richiederebbe circa $10^{154}/10^{19} = 10^{135}$ sec
- 1 anno ha $\leq 10^8$ sec $\Rightarrow > 10^{127}$ anni di calcolo ...

N.B. Esistono algoritmi più efficienti ma sempre con complessità esponenziale.

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

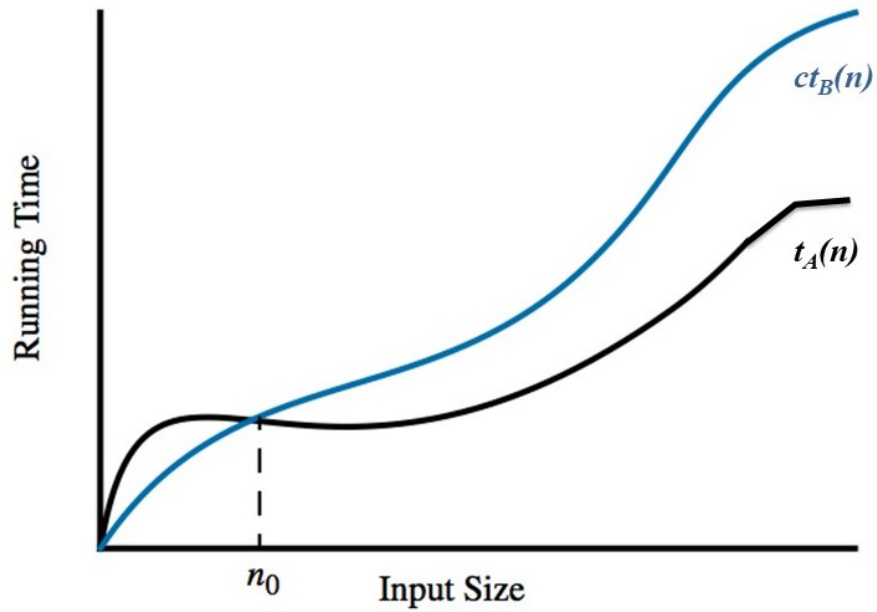
RSA is used in almost all internet-based commercial transactions. Without it, commercial online activities would not be as widespread as they are today. It allows users to communicate sensitive information like credit card numbers over an unsecure internet without having to agree on a shared secret key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

Efficienza asintotica degli algoritmi

A,B che risolvono Π

$t_A(n), t_B(n)$ complessità di A, B (rispettivamente) al caso pessimo

$t_A(n) \in o(t_B(n)) \Rightarrow$ A è "*asintoticamente più efficiente*" di B.



N.B.: 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.

   Cosa fare in questo caso? Dare una stima delle costanti.

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.
   - Aggirare il caso pessimo probabilisticamente.
   - Fare un'analisi al caso medio.

Quando consideriamo alte le costanti?

ESEMPI

(1) $t_A(n) = 10^{100} \cdot n \in \Theta(n)$
$t_B(n) = n^2 \in \Theta(n^2)$
} A è asintoticamente più efficiente di B

Ma $t_A(n) < t_B(n) \Leftrightarrow n > 10^{100}$ (> # atomi dell'universo)

(2) $t_A(n) = 400 \log_2 n \in \Theta(\log n)$
$t_B(n) = (\log_2 n)^2 \in \Theta((\log n)^2)$
} A è asintoticamente più efficiente di B

Ma $t_A(n) < t_B(n) \Leftrightarrow \log_2 n > 400 \Leftrightarrow n > 2^{400} > 10^{100}$

CORRETTEZZA

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 $Q(n_0), Q(n_0+1), \ldots, Q(n_0+k)$
- **PASSO INDUTTIVO**: si fissa un valore $n \geq n_0 + k$ *arbitrario* e si dimostra che

$$Q(m) \text{ vera } \forall m : n_0 \leq m \leq n \Rightarrow Q(n+1) \text{ vera.}$$

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 [GTG14] descrive l'induzione con $n_0 = 1$.

Esempio (Induzione)

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

N.B.: implica che $Q(n) \in \Theta\left(n^2\right)$.

$n_0 = 0 \quad k = 0$

BASE : $Q(0) : \sum_{i=0}^{0} i = 0 = \frac{0 \cdot 1}{2}$ OK

PASSO INDUTTIVO : Fisso $n \geq n_0 + k = 0$ arbitrario

Ip. 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

$$\sum_{i=0}^{n+1} i = n+1 + \sum_{i=0}^{n} i$$

$$= n+1 + \frac{n(n+1)}{2} \quad \text{per ip. induttiva}$$

$$= \frac{(n+1)(n+2)}{2} \Rightarrow Q(n+1) \text{ è vera}$$

Esercizio [GTG14, C-4.35]

Dimostrare per induzione la seguente proprietà:

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

N.B.: implica che $\sum_{i=0}^{n} i^2 \in \Theta\left(n^3\right)$

Per Casa

Osservazione Per k costante, $\sum_{i=0}^{n} i^k \in \Theta(n^{k+1})$

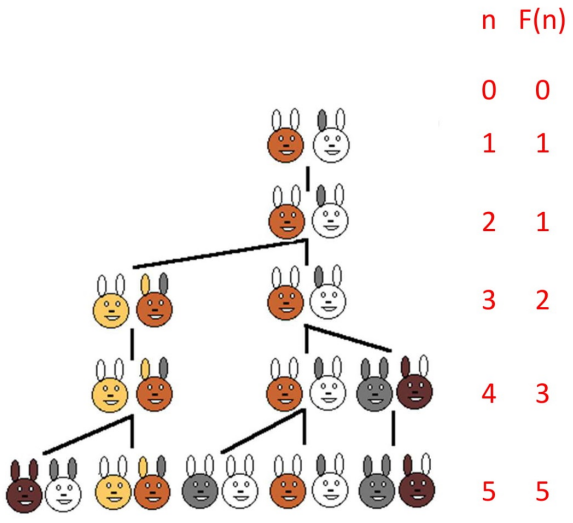
Esempio: successione di Fibonacci

Successione di Fibonacci (in [GTG14] è definita in modo diverso):

- $F(0) = 0$, $F(1) = 1$
- $F(n+1) = F(n) + F(n-1), \forall n \geq 1$

$F(n)$ = 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: una coppia neonata



Esercizio

Dimostrare che

$$F(n) = \frac{1}{\sqrt{5}}\left(\Phi^n - \hat{\Phi}^n\right) \quad \forall n \geq 0$$

$(\Rightarrow F(n) \in \Theta(\Phi^n))$

dove $\Phi = \frac{1+\sqrt{5}}{2}$ (*golden ratio*) e $\hat{\Phi} = \frac{1-\sqrt{5}}{2}$.

$\Phi = 1.618..$

Induzione fallace

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

- $k = 0$
- BASE: $F(0) = 0 \Rightarrow$ OK
- PASSO INDUTTIVO: fissato $n \geq 0$ e assumendo $F(m) = 0$ per ogni $0 \leq m \leq n$ (hp. induttiva), si ha

$$F(n+1) = F(n) + F(n-1) = 0 + 0 = 0.$$

Dov'è l'errore?

Il passo induttivo non è giusto quando $n = 0$ perché $F(n+1) = F(n) + F(n-1)$ vale solo per $n \geq 1$

$\Rightarrow$ In questo caso è necessario usare una base ampia: $n_0 = 0$ $k = 1$

Correttezza

Terminazione

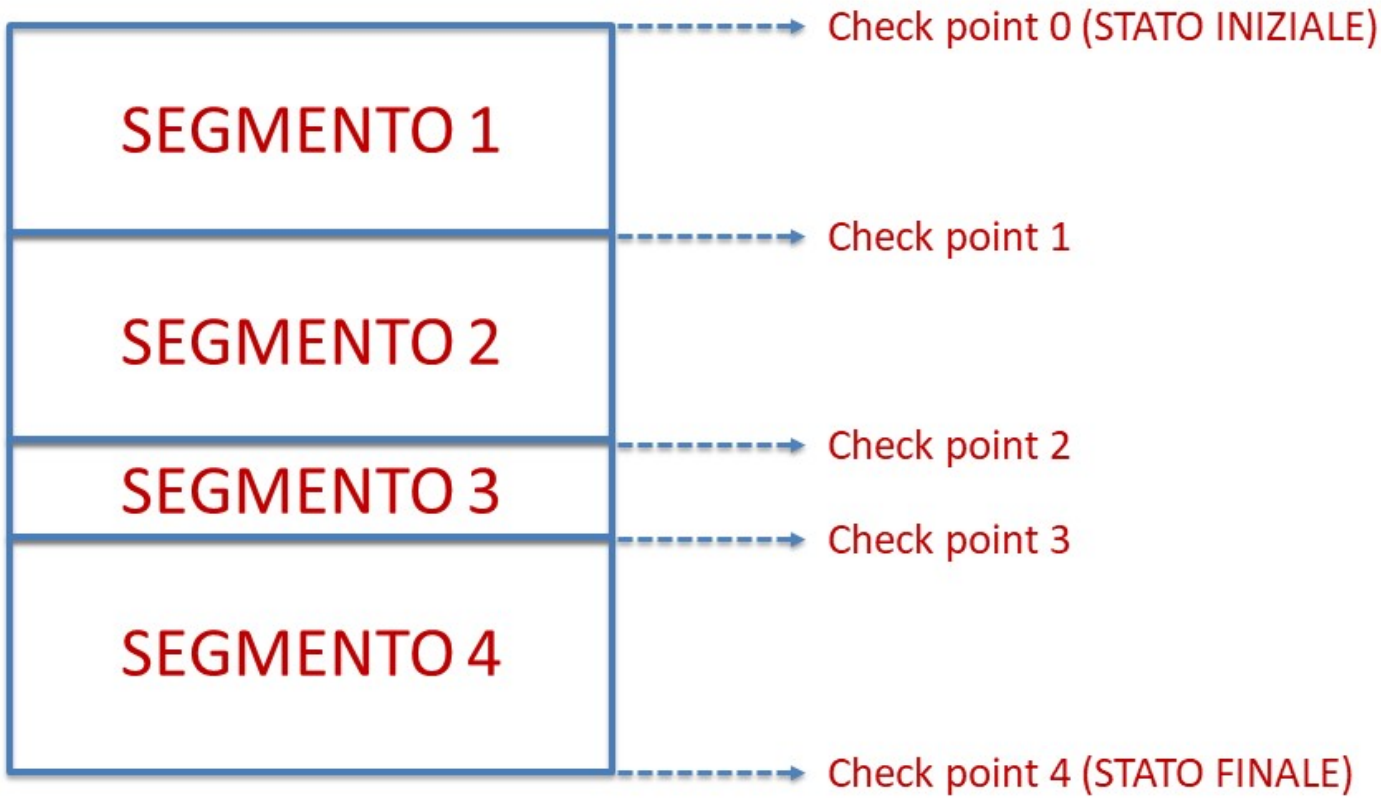
Per provare la terminazione è necessari assicurarsi che i cicli (inclusi i GOTO) e l'eventuale ricorsione abbiano termine.

Soluzione del problema computazionale

Approccio generale:

- Definire lo stato iniziale dell'algoritmo e quello finale che esso deve raggiungere.
- Decomporre `A` in segmenti e definire per ogni segmento lo stato in cui l'algoritmo si deve trovare al 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.

Segmenti notevoli: cicli (for, while, repeat-until)


SEGMENTO 1
SEGMENTO 2
SEGMENTO 3
SEGMENTO 4
Check point 0 (STATO INIZIALE)
Check point 1
Check point 2
Check point 3
Check point 4 (STATO FINALE)

Analisi di cicli

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

Correttezza di un ciclo (for, while, repeat-until ...): provare la correttezza di un ciclo consiste nel 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 tale fine, si fa uso di un **INVARIANTE**

Definizione

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.

N.B. L'invariante va scelto in modo che permetta di dimostrare che alla fine del ciclo vale $\mathcal{L}$

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. Si dimostra in modo induttivo 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 ≥ 0 iterazioni delle istruzioni in esso contenute (*corpo del ciclo*). Ad esempio:

$$\textbf{for } i \leftarrow 1 \textbf{ to } n \textbf{ do } \{\textit{corpo del ciclo}\}$$

è un ciclo che esegue sempre n iterazioni.

Esempio: `arrayMax(A)`

Input: Array $A[0 \div n-1]$ di $n \geq 1$ interi
Output: max intero in A

currMax $\leftarrow A[0]$;
for $i \leftarrow 1$ *to* $n-1$ **do** currMax $\leftarrow$ max { currMax, $A[i]$ };
return currMax

PROPRIETÀ $\mathcal{L}$: Alla fine del ciclo, currMax è il massimo intero in A ($\Rightarrow$ il valore restituito è corretto)

INVARIANTE: Alla fine della iterazione $i \geq 0$

$$currMax = \max\{A[0], A[1], \ldots, A[i]\}$$

Oss. Fine iterazione $i=0$ $\equiv$ inizio del ciclo

① All'inizio del ciclo ($i=0$) l'invariante è reso vero dall'assegnamento $curMax \leftarrow A[0]$

② Supponiamo l'invariante vero a fine iterazione $i<n-1$
$\Rightarrow curMax = max\{A[0], \ldots, A[i]\}$ e dimostriamo che vale anche alla fine della successiva iterazione

Nella iterazione $i+1$, l'istruzione

$curMax \leftarrow max\{curMax, A[i+1]\}$

assicura che alla fine di tale iterazione

$curMax = max\{A[0], \ldots, A[i+1]\}$

$\Rightarrow$ l'invariante continua a valere alla fine della iterazione i+1

(3) Alla fine dell'ultima iterazione ($i = n-1$)

Se vale l'invariante, cioè se

$$curMax = max\{A[0] \dots A[n-1]\}$$

allora curMax è effettivamente il massimo intero in A, che coincide con la proprietà L

Esempio: `arrayFind(A)`

Input: elemento x, array $A[0 \div n-1]$ di n elementi
Output: indice $i \in [0, n)$ t.c. $A[i] = x$, se esiste, altrimenti -1

$i \leftarrow 0$;
while $i < n$ **do**
    if $(x = A[i])$ **then return** i;
    else $i \leftarrow i + 1$;
return -1

PROPRIETÀ $\mathcal{L}$: Il valore restituito è corretto

INVARIANTE : Alla fine di una generica iterazione

$x \neq A[j] \quad \forall 0 \leq j < i$

con i valori delle variabili omonime alla fine della iterazione considerata

① All'inizio del ciclo ($i = 0$) l'invariante vale per VACUITÀ' dato che il range $0..i-1$ è vuoto

② Supponiamo l'invariante vero alla fine di una generica iterazione $\Rightarrow x \neq A[j] \quad \forall 0 \leq j < i < n$

Alla fine della successiva iterazione

* se $x = A[i] \Rightarrow$ i non cambia e il ciclo termina
* se $x \neq A[i] \Rightarrow$ i diventa $i+1$

In entrambi i casi l'invariante continua a valere

③ Fine ultima iterazione: 2 possibili uscite

U1: $x = A[i]$ In questo caso si restituisce i e chiaramente L vale

U2: $i = n$ In questo caso si restituisce -1

Dato che l'invariante mi dice che $x \neq A[j]$

$\forall j : 0 \leq j < n = i$ so che x non è in A

$\Rightarrow$ restituire -1 è corretto $\Rightarrow$ L vale

Esercizio

Sia $S[1 \div n]$ una sequenza di n bit. Il seguente algoritmo determina la lunghezza del più lungo segmento continuo di 1.

Algoritmo `maxSegment1`

Input: Sequenza S di n bit
Output: Lunghezza del più lungo segmento continuo di 1

```
max ← 0; curr ← 0;
for i ← 1 to n do
    if S[i] = 1 then
        curr ← curr +1;
        if curr > max then max ← curr;
    else curr ← 0;
return max
```

Dimostrare la correttezza del ciclo (ovvero dell'algoritmo) tramite un opportuno invariante.

Esercizio

Dimostrare la correttezza dell'algoritmo iterativo per la ricerca binara richiesto dal secondo esercizio della Slide 16

Esercizio

Sia A una matrice $n \times n$ di interi, con $n \geq 2$, dove righe e colonne sono numerate a partire da 0.

1. Sviluppare un algoritmo che restituisce l'indice $j \geq 0$ di una colonna di A con tutti 0, se tale colonna esiste, altrimenti restituisce -1. L'algoritmo deve contenere un solo ciclo.
2. Trovare un upper bound e un lower bound alla complessità dell'algoritmo sviluppato.
3. Provare la correttezza dell'algoritmo sviluppato, scrivendo un opportuno invariante per il ciclo su cui esso si basa.

Esercizio

Sia A una matrice binaria $n \times n$ per la quale valgono le seguenti proprietà: (a) in ciascuna riga gli 1 vengono prima degli 0; e (b) il numero di 1 nella riga i è $\leq$ del numero di 1 nella riga $i+1$, per $0 \leq i \leq n-2$. Descrivere un algoritmo che in tempo $O(n)$ conti il numero di 1 in A, e provarne la correttezza. L'algoritmo deve contenere un solo ciclo.

Ricorsione

Algoritmo Ricorsivo: algoritmo che invoca se stesso (su istanze sempre più piccole) sfruttando la nozione di induzione.

La soluzione di una istanza di taglia n è ottenuta:

- direttamente: se $n = n_0, n_0 + 1, \ldots, n_0 + k$ (*casi base*) $k \geq 0$
- riconducendosi alla soluzione di $r \geq 1$ istanze di taglia $< n$: se $n > n_0 + k$. Se $r = 1$, si parla di *linear recursion*.

ESEMPI:

- Algoritmi `linearSum` e `reverseArray` (entrambi esempi di linear recursion) illustrati nelle prossime slide.
- Algoritmo `MergeSort`.

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

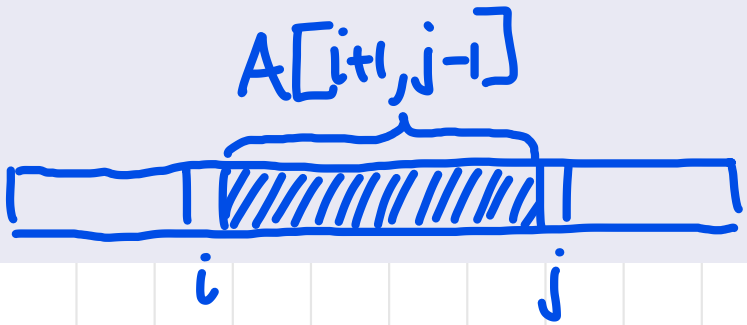
CASO BASE : n = 1 $(n_0 = 1, k = 0)$

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 $n \leq 1$

Per ribaltare tutto A, la prima invocazione è

reverseArray $(A, 0, |A|-1)$

Esecuzione di un algoritmo ricorsivo

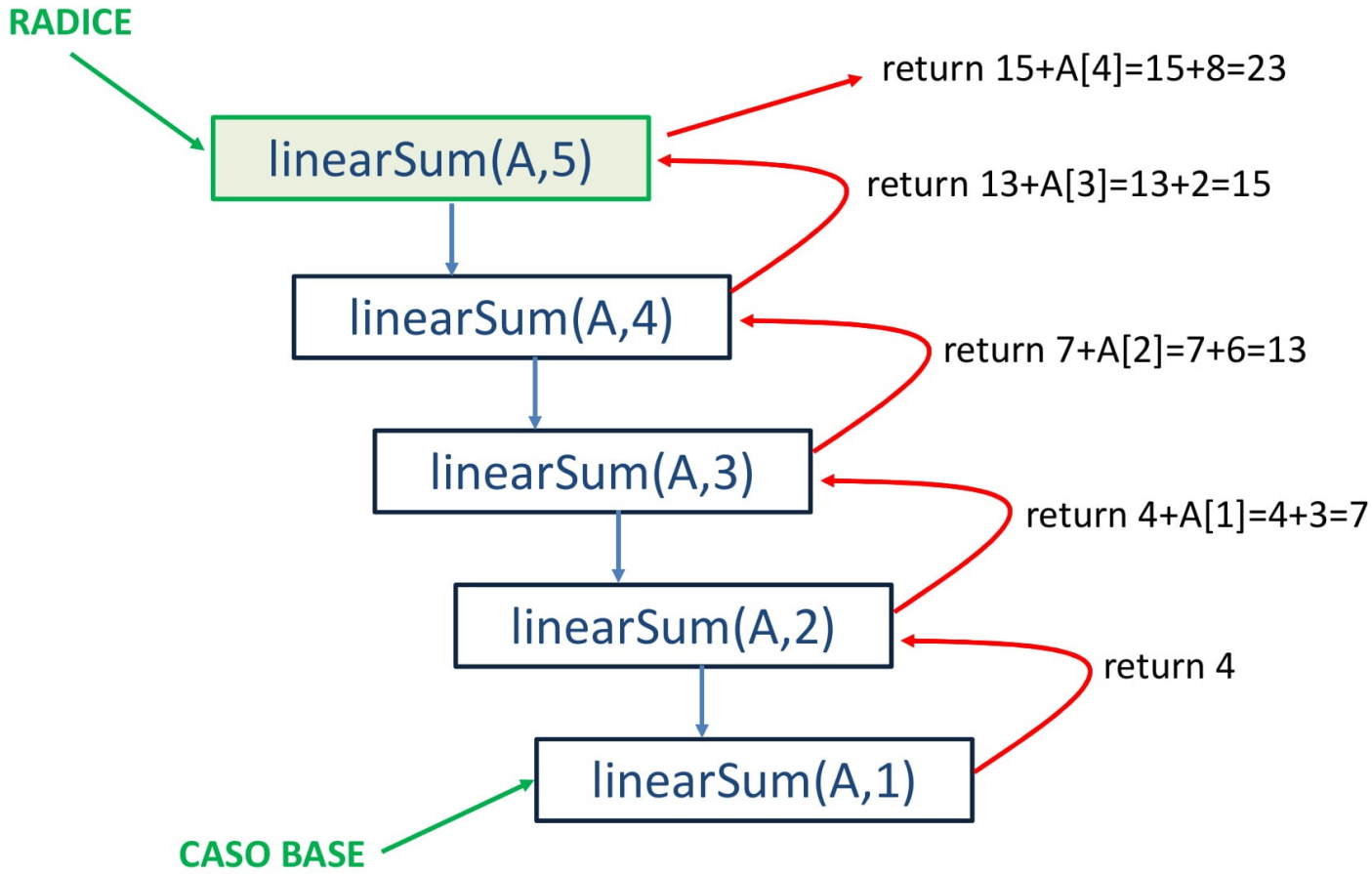
Albero della ricorsione

All'esecuzione di un algoritmo ricorsivo su una data istanza è associato un albero della ricorsione tale che:

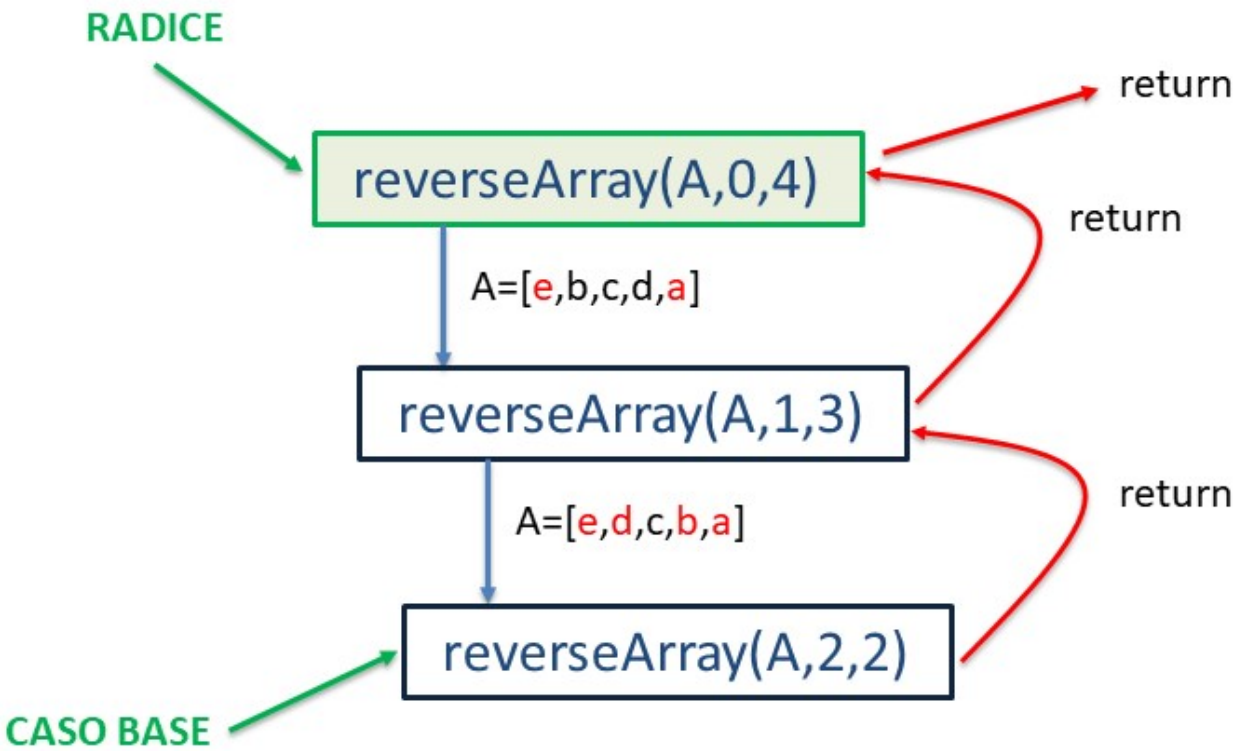
- Ogni nodo corrisponde a un'invocazione ricorsiva distinta fatta durante l'esecuzione dell'algoritmo.
- La radice dell'albero corrisponde alla prima invocazione, e i figli di un nodo x sono associati alle invocazioni ricorsive fatte direttamente dall'invocazione corrispondente a x.
- Le foglie dell'albero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

N.B. [GTG14] chiama l'albero della ricorsione *recursion trace*.

Albero della Ricorsione per linearSum(A,5) con A=[4,3,6,2,8]



Albero della Ricorsione per reverseArray(A,0,4) con A=[a,b,c,d,e]



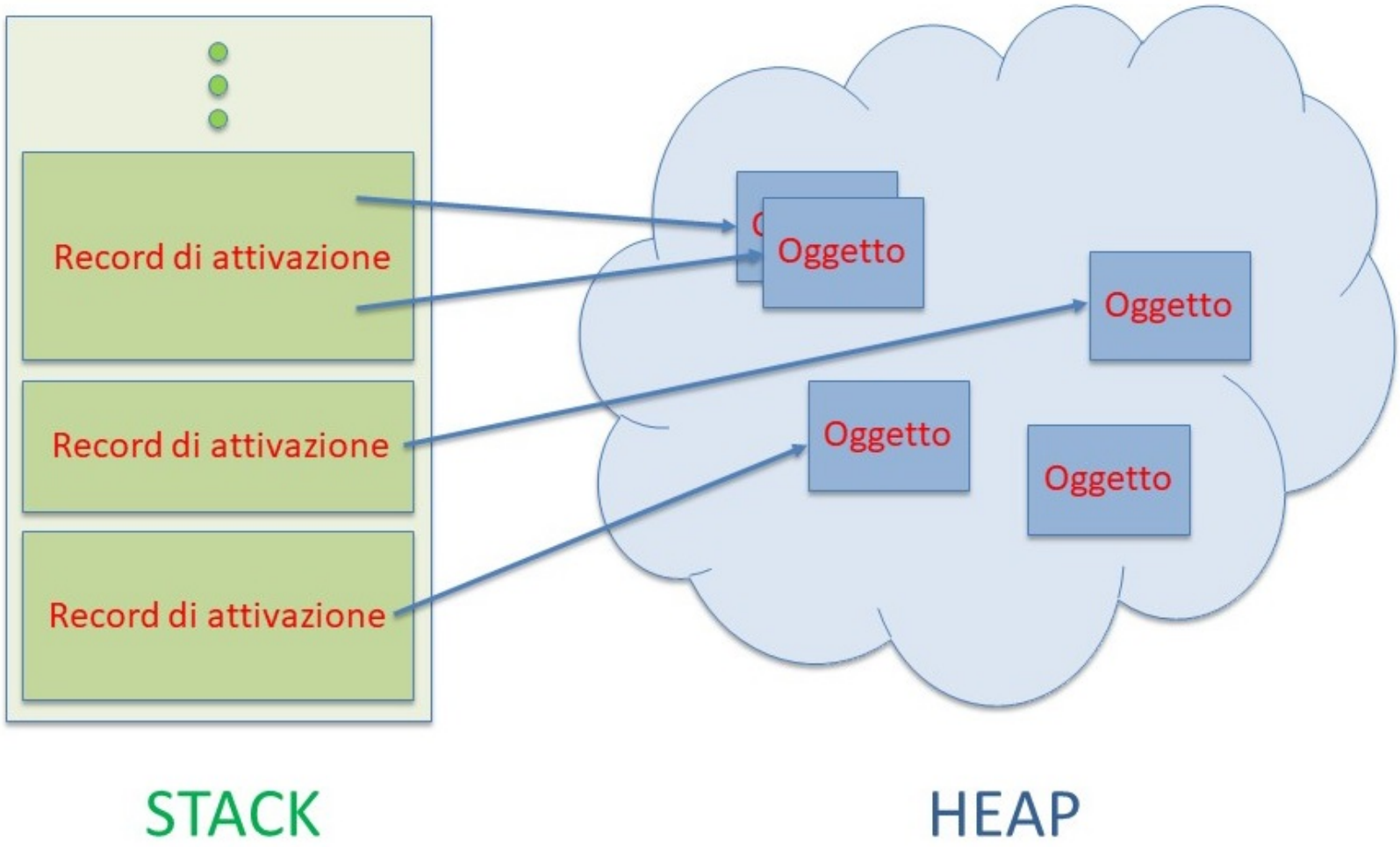
Alla fine dell'ultima invocazione si ha A=[e,d,c,b,a]

Esecuzione di un algoritmo ricorsivo

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

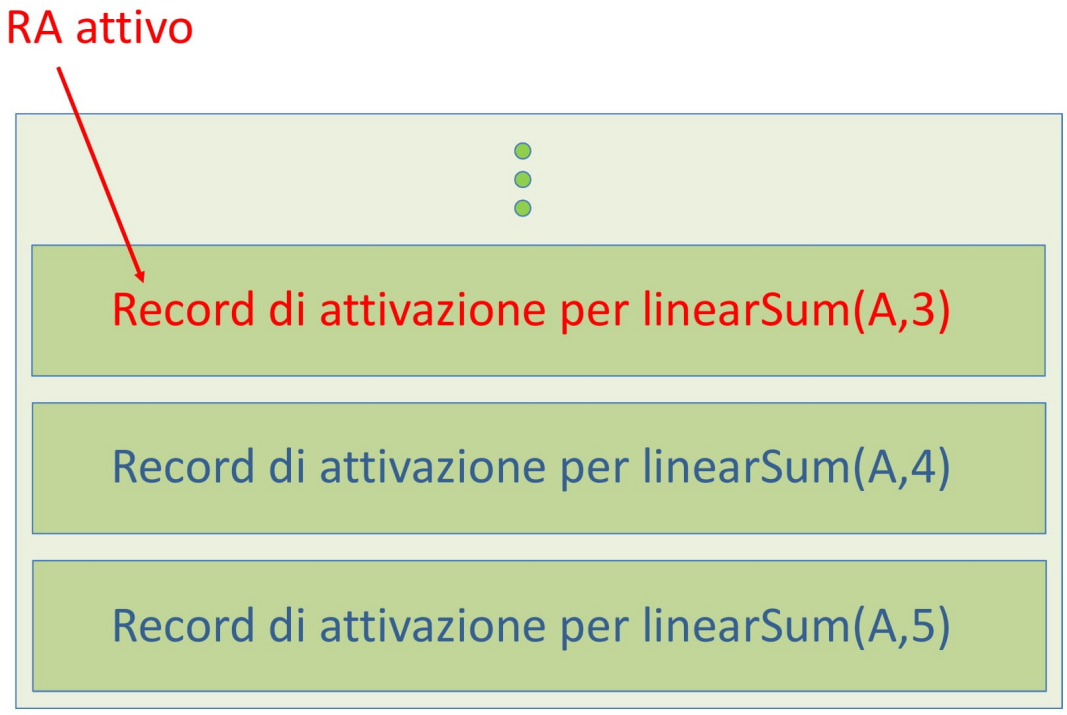
- STACK: spazio destinato a memorizzare variabili locali ai metodi e riferimenti a oggetti.
  - Per ogni invocazione di un metodo viene inserito un record di attivazione (RA) contenente variabili e riferimenti a oggetti relativi a quella invocazione. (In inglese si usa il termine *activation record* o *activation frame*.)
  - Un RA viene eliminato quando l'invocazione di metodo corrispondete finisce l'esecuzione.
  - I RA sono inseriti/eliminati con politica LIFO, e 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.

Esecuzione di un algoritmo ricorsivo



Esecuzione di un algoritmo ricorsivo

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



Algoritmi ricorsivi vs algoritmi iterativi

- **Algoritmo ricorsivo:** algoritmo che invoca se stesso su istanze più piccole
- **Algoritmo iterativo:** algoritmo non-ricorsivo

Osservazione: il termine *iterativo* è usato per evidenziare che (tranne 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 (ad es. `linearSum`, `reverseArray`) ma può anche averli (ad es., `MergeSort`).

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 dalla 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 va bene per tutte le istanze.

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`$(A, n-1) + A[n-1]$;

Albero della ricorsione associato a una generica istanza di taglia n

* n nodi associati a linearSum(A, j) $j = n, n-1, \ldots, 1$
* Costo associato a ciascun nodo: $\Theta(1)$ operazioni (esclude quello delle chiamate ricorsive al suo interno)

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

$\Rightarrow$ $t_{\text{ricorsiva}}(n) \in \Theta(n)$

↓

complessità al caso pessimo

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

N.B. La taglia dell'istanza è $n = j-i+1$

Albero della ricorsione associato a una generica istanza di taglia n $(n = j-i+1)$

* $\lfloor n/2 \rfloor + 1$

* Costo associato a ciascun nodo: $\Theta(1)$ operazioni

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

$\Rightarrow$ $t_{reverseArray}(n) \in \Theta(n)$

Calcolo efficiente di potenze

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

Osservazione chiave

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

Algoritmo `power`(x, n)

Input: $x \in \mathbb{R}$ e $n \geq 0$ intero
Output: $p(x, n)$
if $(n = 0)$ **then return** 1 (CASO BASE);
if $(n$ è dispari$)$ **then**
    $y \leftarrow$`power`$(x, (n-1)/2)$;
    return $x \cdot y \cdot y$;
else
    $y \leftarrow$`power`$(x, n/2)$;
    return $y \cdot y$

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 $n_i \leq n/2^i$. 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 $\Theta(1)$.

$\Rightarrow t_{\texttt{power}}(n) \in O(\log n)$

Applicazione di power al calcolo di $F(n)$

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

Algoritmo `powerFib`(n)
Input: intero $n \geq 0$
Output: $F(n)$
$\Phi \leftarrow ((1+\sqrt{5})/2)$;
$\hat{\Phi} \leftarrow ((1-\sqrt{5})/2)$;
return $(\texttt{power}(\Phi, n) - \texttt{power}(\hat{\Phi}, n))/\sqrt{5}$

NON È UN ALGORITMO RICORSIVO
Ma usa un algoritmo ricorsivo al suo interno

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

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:

- Si dimostra la correttezza per i casi base $n \in [n_0, n_0 + k]$
- 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 $n + 1$

Esempio: correttezza di `linearSum`(A, n) per $n \geq 1$

CASO BASE ($n=1$): corretto banale

PASSO INDUTTIVO: Fisso $n \geq 1$ arbitrario

Hp. Induttiva: linearSum(A, j) è 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

linearSum$(A, n) + A[n]$

$$= \left(\sum_{i=0}^{n-1} A[i] \right) + A[n] \quad \text{per Hp. induttiva}$$

$$= \sum_{i=0}^{n} A[i]$$

$\Rightarrow$ Il valore restituito è corretto $\square$

Esercizi

Esercizio

Provare la correttezza di `reverseArray` e `power`.

Esercizio C-5.10 del testo [GTG14]

Il problema della *element uniqueness*, definito a pagina 162 del testo [GTG14], chiede che dato un array A di n elementi si determini se tali elementi sono tutti distinti.

1. Progettare un algoritmo ricorsivo `EU` efficiente per risolvere questo problema, senza fare ricorso all'ordinamento.
2. Analizzare la complessità $t_{\mathrm{EU}}(n)$ usando l'albero della ricorsione.

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 pessimo di un algoritmo
  - Definizione
  - Analisi asintotica espressa tramite ordini di grandezza ($O(), \Omega(), \Theta()$).
- 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

Per chi vuole mettersi alla prova ...

Esercizio

Sia A un array di n interi arbitrari. Progettare e analizzare un algoritmo che trova, se esiste, l'intero che occorre in A almeno $\lfloor n/2 \rfloor + 1$ volte (*maggioranza assoluta*). L'algoritmo deve avere complessità $O(n)$.

- Versione 1: assumere A ordinato (facile)
- Versione 2: nessuna assunzione su A (difficile)