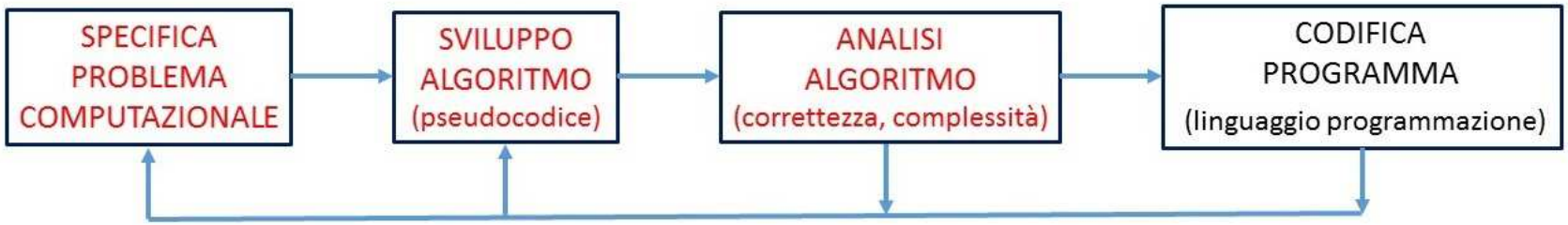
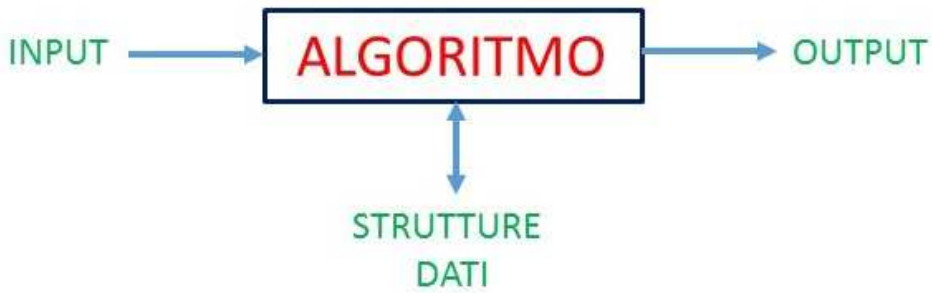
Dati e Algoritmi 1

Andrea Pietracaprina

Ingegneria dell'Informazione

Introduzione al corso





Introduzione al corso

Definire problemi computazionali e progettare algoritmi e strutture dati efficienti è fondamentale per moltissime applicazioni attuali:

- Big data: estrarre informazioni da grandi quantità di dati
- Genomica
- Analisi di reti
  - telecomunicazioni: wireless, sensor networks, internet of things
  - biologia: PPI networks
  - web e reti sociali (facebook, twitter, ecc.)
  - reti elettriche (power grids)

Obiettivi formativi

- Capacità di progetto e di analisi di algoritmi/strutture dati
- Conoscenza di building block algoritmici fondamentali
- Capacità di problem solving
- Rigore nel ragionamento e nel linguaggio: *efficace*, *essenziale*, *non ambiguo*, *senza salti logici*

Aspetti organizzativi

- Iscrizione al corso (su Elearning)
  pass: INF1920
- Strumenti online:
  - **Elearning:** iscrizione, forum, risultati esami
  - **Uniweb:** liste d'esame e pubblicazione voti
  - **Sito del corso:** *http://www.dei.unipd.it/∼capri/DA1/*
    materiale e info complete

Argomenti

- Nozioni di base (12h)
- Text processing (6h)
- Ripasso di Java (4h)
- Alberi (10h)
- Priority queue (10h)
- Mappe e dizionari (10h)
- Grafi (10h)
- Ordinamento (10h)

Materiale

- **Lucidi:** resi disponibili in anticipo e a blocchi
- **Dispense (di esercizi):** rese disponibili sul sito del corso
- **Appunti:** da prendere a lezione
- **Testo:** [GTG14] M.T. Goodrich, R. Tamassia, M.H. Goldwasser. Data Structures and Algorithms in Java, 6/e John Wiley and Sons, 2014.

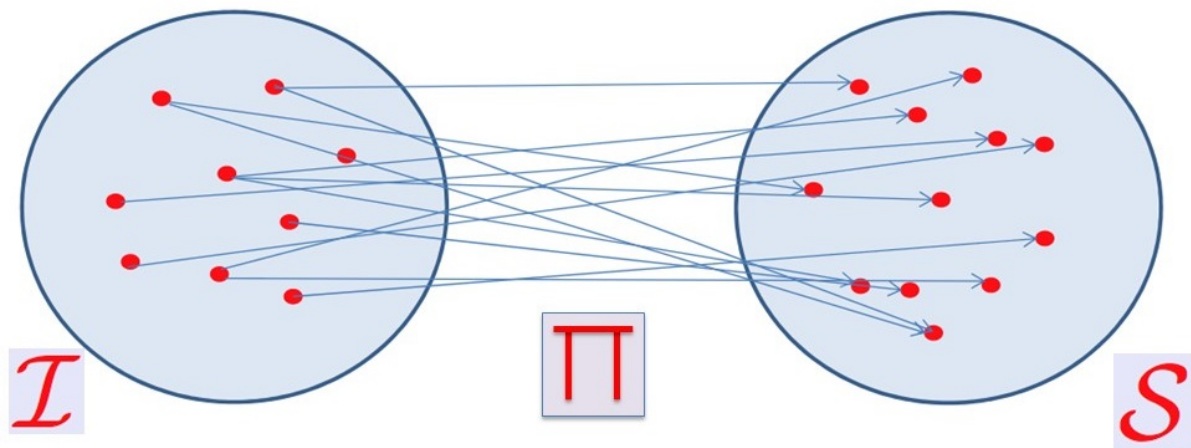
Nozioni di base

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* Π mette in relazione un insieme $\mathcal{I}$ di possibili input (chiamati *istanze*) con un insieme $\mathcal{S}$ di possibili output (chiamati *soluzioni*). Precisamente, Π è *sottoinsieme del prodotto cartesiano* $\mathcal{I} \times \mathcal{S}$, cioè un insieme di coppie (i, s) con $i \in \mathcal{I}$ e $s \in \mathcal{S}$. Per concretezza, Si richiede che per ogni istanza $i \in \mathcal{I}$ esistano ≥ 1 soluzioni $s \in \mathcal{S}$ tali che $(i, s) \in \Pi$.



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.

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

Esempi

Ordinamento di sequenze 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.

$$(<43,16,75,2>,<4,2,1,3>) \in \Pi$$
$$(<7,1,7,3,3,5>,<2,4,5,6,1,3>) \in \Pi$$
$$(<7,1,7,3,3,5>,<2,5,4,6,1,3>) \in \Pi$$
$$(<13,4,25,17>,<1,2,4,3>) \ ? \ ?$$

Osservazioni

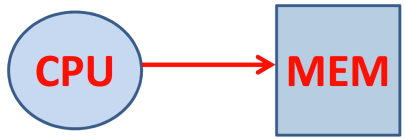
- una soluzione può essere associata a più istanze diverse (ad es., somma di interi)
- un'istanza può avere più soluzioni (ad es., ordinamento ver. 2)

Algoritmo e modello di calcolo

Definizione

Un *algoritmo* è una procedura computazionale ben definita che transforma un dato *input* in un *output* eseguendo una sequenza finita di *passi elementari*. L'algoritmo fa riferimento a un *modello di calcolo* (o *modello di computazione*), ovvero un'astrazione di computer che definisce l'insieme di passi elementari.

Il modello di calcolo più utilizzato è il modello *RAM* [1] (*Random Access Machine*)



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

[1] Diverso da *Random Access Memory*!

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

1. `A` riceve come input istanze $i \in \mathcal{I}$
2. `A` produce come output soluzioni $s \in \mathcal{S}$
3. Dato un input $i \in \mathcal{I}$, `A` produce come output $s \in \mathcal{S}$ tale che $(i, s) \in \Pi$

In altre parole: `A` calcola una funzione che mappa ogni istanza in una soluzione di tale istanza. (Più soluzioni? `A` ne calcola una.)

Per semplicità e facilità di analisi, un algoritmo viene di solito specificato tramite *pseudocodice*, ovvero un mix di costrutti di linguaggi di programmazione ad alto livello e linguaggio naturale.

Esempio di pseudocodice

Algoritmo `Transpose(A)`
Input: matrice A $n \times n$
Output: matrice trasposta A^T
$i \leftarrow 0; j \leftarrow 1;$
while $i < n - 1$ **do**
    scambia $A[i, j]$ *e* $A[j, i]$;
    if $j = n - 1$ **then** $i \leftarrow i + 1; j \leftarrow i + 1;$
    else $j \leftarrow j + 1;$
return A

Osservazioni

L'uso dello pseudocodice

- Assume un implicito accordo sulle operazioni che possono essere considerate come "passi elementari"
- Deve garantire che la sequenza di passi elementari eseguiti per ogni input sia facilmente ricavabile

Esercizio

Siano A e B due array di n interi ciascuno. Scrivere un algoritmo in pseudocodice per calcolare un array C di n interi, tale che

$$C[i] = A[i] \cdot B[i].$$

per ogni $0 \leq i \leq n-1$.

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 maniera crescente, ed x un valore intero. Scrivere un algoritmo in pseudocodice che restituisca l'indice di x in A, se x appare in A, e -1 altrimenti (l'algoritmo deve implementare la ricerca binaria).

Per *taglia (size) di un'istanza* si intende una funzione che mappa ogni istanza in uno (o più) valori, i quali forniscono una misura della sua grandezza.

La taglia è utilizzata per partizionare l'universo delle istanze in sottoinsiemi costituiti da istanze tra loro *simili* e *confrontabili*, in modo che l'analisi di un algoritmo possa essere espressa parametricamente in essa.

Ad esempio, l'affermazione *l'algoritmo MergeSort richiede tempo* $O(n \log n)$ specifica la complessità dell'algoritmo in funzione del numero n di elementi da ordinare, cioè della taglia dell'istanza del problema di ordinamento che `MergeSort` risolve.

Strutture dati

Gli algoritmi utilizzano strutture dati per organizzare e accedere in modo sistematico ai dati di input e a dati intermedi generati durante l'esecuzione.

Definizione

Una *struttura dati* è una collezione di *oggetti* corredata di *metodi* per accedere e/o modificare la collezione. Nella definizione di una struttura dati si distinguono due *livelli di astrazione*:

1. *livello logico*: specifica l'organizzazione logica degli oggetti della collezione, e la relazione input-output che caratterizza ciascun metodo
2. *livello fisico*: specifica il layout fisico dei dati e l'implementazione dei metodi tramite opportuni algoritmi

La specifica al livello logico di una struttura dati viene riferita come *Abstract Data Type* (*ADT*). In *Java*, la specifica a livello logico di una struttura dati è fatta da (una gerachia di) *interfacce*, mentre la specifica a livello fisico è fatta da (una gerachia di) *classi*.

Esercizio

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

Risoluzione

- $\mathcal{I} = \{X \ : \ X = \text{stringa binaria}\}$;
- $\mathcal{S} = \{(j,k) \ : \ j,k \in Z \text{ con } j \geq -1, k \geq 0\}$;
- $\Pi =$ insieme delle coppie $(X,(j,k))$ con $X \in \mathcal{I}$ e $(j,k) \in \mathcal{S}$ tali che:
  - Se X ha tutti 0, allora $j = -1$ e $k = 0$
  - Altrimenti, il più lungo segmento di 1 consecutivi inizia all'indice j ed è lungo k ($X[j \div j + k - 1]$)

Esercizio

Specificare come problema computazionale Π la verifica se due insieme finiti di oggetti da un universo $\mathcal{U}$ sono disgiunti oppure no.

Risoluzione

- $\mathcal{I} = \{(X, Y) \ : \ X, Y \subseteq U\}$;
- $\mathcal{S} = \{\texttt{yes}, \texttt{no}\}$;
- $\Pi =$ insieme delle coppie $((X, Y), s)$ con $(X, Y) \in \mathcal{I}$ e $s \in \mathcal{S}$ tali che:
  - Se $X \cap Y = \emptyset$ allora $s = \texttt{yes}$
  - Se $X \cap Y \neq \emptyset$ allora $s = \texttt{no}$.

Caso di studio 1: Google News

PROBLEMA (*Diversity Maximization*): Trovare un insieme di k articoli *molto diversi* tra quelli presenti su siti di giornali, tv, radio, ecc. Per formalizzare il problema si fanno prima alcune scelte:

- Si sceglie un'opportuna rappresentazione degli articoli come vettori in $\Re^D$, per un qualche D (ne esistono diverse)
- Si sceglie una funzione distanza tra articoli, ovvero una funzione $d : \Re^D \times \Re^D \to \Re$ (ad es., distanza Euclidea)
- Si sceglie una funzione *diversity*, basata su d, che associa a ciascun insieme $Y \subset \Re^D$ un valore reale. Ad es.:

$$\text{diversity}(Y) = \min_{u,v \in Y} d(u,v).$$

Fatte le scelte sopra indicate, il problema computazionale diventa:

- $\mathcal{I} = \{(X,k) : X \subset \Re^D, k \text{ intero in } [1, |X|]\}$;
- $\mathcal{S} = \{Y : Y \subset \Re^D\}$;
- $\Pi = \{((X,k), Y) : (X,k) \in \mathcal{I}, Y \in \mathcal{S}$ e Y è il sottoinsieme di X di taglia k che massimizza $\text{diversity}(Y)\}$.

Esercizi

Esercizio

Sia T una stringa arbitraria di lunghezza $n \geq 1$ su un alfabeto Σ. È sempre possibile scrivere T come concatenazione di n/m copie di una stringa P (ovvero $T = PP \cdots P$ n/m volte) dove P ha lunghezza $m \leq n$ ed n/m è intero. La *periodicità* di T è il minimo valore m per cui tale scrittura di T è possibile. Ad esempio, $T =$ `abcd` ha periodicità 4 mentre $T =$ `abababab` ha periodicità 2. Definire come *problema computazionale* Π il problema di trovare la periodicità di una stringa T specificando l'insieme delle istanze $\mathcal{I}$, l'insieme delle soluzioni $\mathcal{S}$ e le coppie che costituiscono Π.

Esercizio

Progettare e scrivere in pseudocodice un algoritmo che risolva il problema computazionale "Diversity Maximization" formalizzato nel caso di studio.

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 [GTG14, Par. 4.1]

Obiettivo: stimare il tempo di esecuzione (*running time*) di un algoritmo per valutarne l'efficienza e poterlo confrontare con altri algoritmi per lo spesso problema computazionale.

Osservazioni

Il tempo di esecuzione (di un programma) dipende da

- istanza di input: di solito cresce con la taglia, ma a parità di taglia, input diversi possono avere tempi anche molto diversi (e.g., `InsertionSort`)
- ambiente HW (e.g., processore, sistema di memoria, ecc.)
- ambiente SW (e.g., linguaggio di programmazione, OS, compilatore, etc.)

Come possiamo studiare la complessità in tempo?

Studio Sperimentale

In `Java`: uso `System.currentTimeMillis()`

- ritorna il numero (`long`) di millisecondi trascorsi da un evento di riferimento (la mezzanotte del 1/1/1970)
- invocazione: prima e dopo esecuzione algoritmo $\Rightarrow$ differenza

Buona idea? OK ma con dei limiti

- non può considerare tutti gli input
- richiede l'implementazione di un algoritmo con un programma
  - molto lavoro!
  - confronto tra algoritmi: difficile (implementazione ha un ruolo)
- HW/SW dependent

Requisiti per l'analisi della complessità in tempo

1. deve considerare tutti gli input
2. deve permettere di confrontare algoritmi (senza necessariamente determinare il tempo di esecuzione esatto)
3. deve essere eseguibile a partire da una specifica *high level* dell'algoritmo ($\Rightarrow$ pseudocodice)

Approccio

- analisi al *caso pessimo* (*worst-case*) in funzione della taglia dell'istanza (req. 1,2)
  - Altri tipi di analisi sono possibili: analisi al caso medio (*average case*), analisi probabilistica
- conteggio passi elementari nel modello RAM (req. 2,3)
- analisi asintotica (per semplificare il conteggio)

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 che $\mathtt{A}$ esegue per risolvere una istanza di taglia n* (operazioni = passi elementari del modello RAM) .

In altre parole:

- Sia $t_{\mathtt{A},i}$ = numero di operazioni eseguite da $\mathtt{A}$ per l'istanza i
- Allora $t_{\mathtt{A}}(n) = \max\{t_{\mathtt{A},i} \ : \ \text{istanza } i \in \mathcal{I} \text{ di taglia } n\}$.

Osservazioni

Si assume che per ogni data istanza, l'algoritmo esegua *una sola* sequenza operazioni. Si parla in questo caso di *algoritmo deterministico*. Più avanti vedremo che esistono algoritmi (*algoritmi probabilistici*) per i quali tale assunzione non vale.

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

È ragionevole considerare n come taglia dell'istanza.

Nel determinare la complessità al caso pessimo di `ArrayMax` incontriamo le seguenti difficoltà:

- Identificare l'istanza peggiore per ogni taglia fissata n.
- Contare in modo preciso il numero di operazioni (passi elementari) eseguiti per risolvere l'istanza peggiore (si noti che la caratterizzazione di "passo elementare" è volutamente lasciata al buon senso).

Limiti inferiori e superiori

Per superare la difficoltà di identificare l'istanza peggiore si ricorre alla stima di limiti superiori/inferiori.

Si cercano funzioni $f_1(n)$ e $f_2(n)$ tali che

- $t_{\mathtt{A}}(n) \leq f_1(n)$ (*limite superiore*)
- $t_{\mathtt{A}}(n) \geq f_2(n)$ (*limite inferiore*)

senza dover necessariamente identificare l'istanza peggiore.

Analisi di `ArrayMax`

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.

Questo implica che esistono costanti $c_1, c_2, c_3, c_4 > 0$ tali che per ogni n

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

Analisi di `ArrayMax`

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

No, perché:

- È difficile contare in modo preciso le operazioni.
- La scelta del set di operazioni è arbitraria e, in pratica, può variare da computer a computer.
- Il tempo di esecuzione, che la complessità vuole stimare, dipende da tanti fattori che è impossibile quantificare in modo preciso, e quindi una quantificazione della complessità precisa sino alle costanti non ha senso

$\Rightarrow$ Si ricorre allora all'**analisi asintotica**

Analisi asintotica

- Si ignorano fattori moltiplicativi costanti (= non dipendenti dalla taglia dell'istanza)
- Si ignorano termini additivi non dominanti

Nel caso di `ArrayMax` è sufficiente stabilire che

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

In questo caso otteniamo una stima stretta (tight) della complessità:

$$t_{\texttt{ArrayMax}}(n) \text{ è proporzionale a } n$$

Per esprimere efficacemente affermazioni come quelle fatte sopra si usano gli **ordini di grandezza**

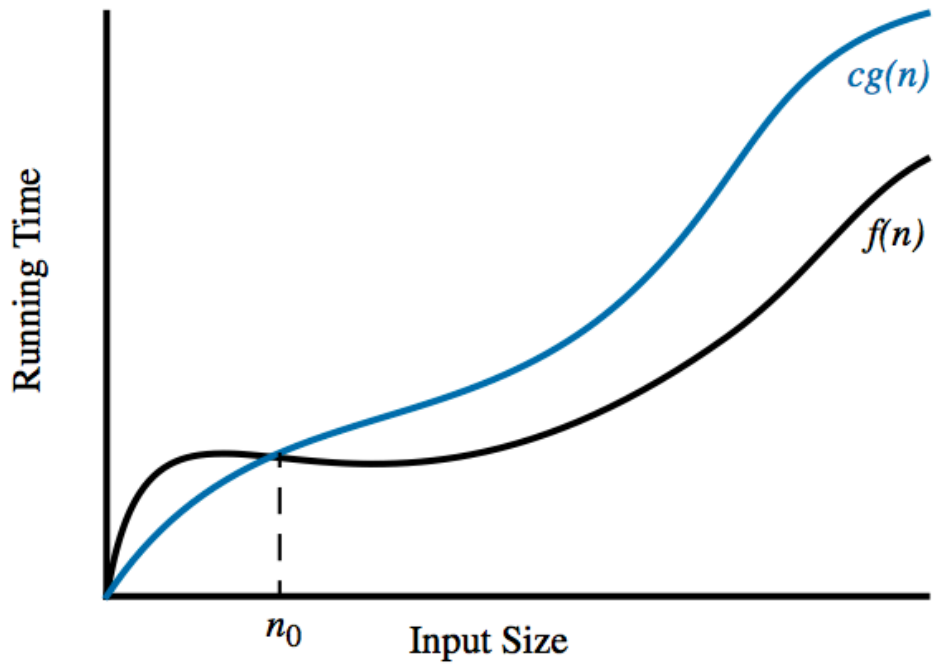
Ordini di grandezza

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

Definizione

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

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



Esempi

- $f(n) = 3n + 4$ per $n \geq 1 \Rightarrow f(n) \in O(n)$ $(c = 7, n_0 = 1)$
- $f(n) = 3n + 4$ per $n \geq 1 \Rightarrow f(n) \in O(n)$ $(c = 4, n_0 = 4)$
- $f(n) = n + 2n^2$ per $n \geq 1 \Rightarrow f(n) \in O\left(n^2\right)$ $(c = 3, n_0 = 1)$
- $f(n) = 3n + 4$ per $n \geq 1 \Rightarrow f(n) \in O\left(n^2\right)$ $(c = 4, n_0 = 4)$
- $f(n) = 2^{100}$ per $n \geq 1 \Rightarrow f(n) \in O(1)$ $(c = 2^{100}, n_0 = 1)$

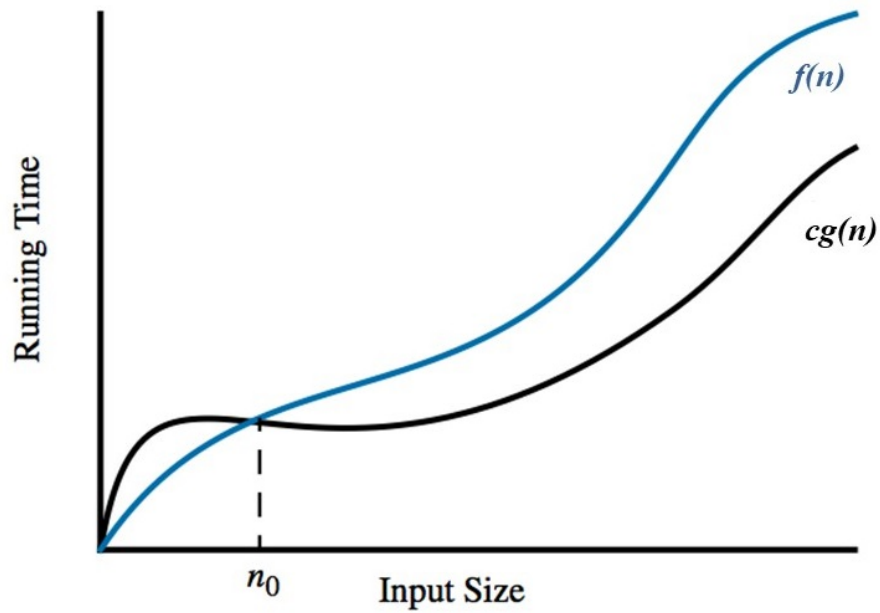
Ricordando che $t_{\texttt{ArrayMax}}(n) \leq c_1 n + c_2$ per $n \geq 1$ e $c_1, c_2 > 0$ costanti, possiamo concludere che

$$t_{\texttt{ArrayMax}}(n) \in O(n) \quad (c = c_1 + c_2, n_0 = 1)$$

Definizione

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

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



Esempi

- $f(n) = 3n + 4$ per $n \geq 1 \Rightarrow f(n) \in \Omega(n)$ $(c = 3, n_0 = 1)$
- $f(n) = 3n + 4$ per $n \geq 1 \Rightarrow f(n) \in \Omega(n)$ $(c = 1, n_0 = 1)$
- $f(n) = n + 2n^2$ per $n \geq 1 \Rightarrow f(n) \in \Omega\left(n^2\right)$ $(c = 1, n_0 = 1)$
- $f(n) = 6n^2 + 1$ per $n \geq 1 \Rightarrow f(n) \in \Omega(n)$ $(c = 6, n_0 = 1)$
- $f(n) = 2^{100}$ per $n \geq 1 \Rightarrow f(n) \in \Omega(1)$ $(c = 1, n_0 = 1)$

Ricordando che $t_{\texttt{ArrayMax}}(n) \geq c_3 n + c_4$ per $n \geq 1$ e $c_3, c_4 > 0$ costanti, possiamo concludere che

$$t_{\texttt{ArrayMax}}(n) \in \Omega(n) \quad (c = c_3, n_0 = 1)$$

Definizione

$f(n) \in \Theta(g(n))$ se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$, ovvero $\exists c', c'' > 0$ e $n_0 \geq 1$, *costanti non dipendenti da* n, tali che:

$$c'g(n) \leq f(n) \leq c''g(n), \forall n \geq n_0$$

Esempi

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

Definizione

$f(n) \in o(g(n))$ se $\forall c > 0\ \exists n_0 \geq 1$, con c e n_0 *costanti non dipendenti da* n, tale che

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

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

N.B. n_0 è in genere una funzione di c.

Esempi

- $f(n) = 100n$ per $n \geq 1 \Rightarrow f(n) \in o\left(n^2\right)$
  (per ogni $c > 0$ si scelga $n_0 = 100/c$)
- $f(n) = 3n/(\log_2 n)$ per $n \geq 1 \Rightarrow f(n) \in o(n)$
  (per ogni $c > 0$ si scelga $n_0 = \left\lceil 2^{\frac{3}{c}} \right\rceil$)

Proprietà degli ordini di grandezza

Proposizione

Si consideri il polinomio $\sum_{i=0}^{k} a_i n^i$ con $a_k > 0$ e k, a_i costanti, e $k \geq 0$. Allora $\sum_{i=0}^{k} a_i n^i \in \Theta\left(n^k\right)$.

Dimostrazione

- $O\left(n^k\right)$: $c = \sum_{i=0}^{k} |a_i|, n_0 = 1$
- $\Omega\left(n^k\right)$: $c = \frac{a_k}{2}, n_0 = \lceil \frac{2k\gamma}{a_k} \rceil$, con $\gamma = \max\{|a_i|, 0 \leq i < k\}$

  Verifica: $\sum_{i=0}^{k} a_i n^i = a_k n^k + \sum_{i=0}^{k-1} a_i n^i \geq a_k n^k - k\gamma n^{k-1}$

  per $n \geq n_0$: $a_k n^k - k\gamma n^{k-1} = \frac{a_k}{2} n^k \left(2 - \frac{2k\gamma}{a_k n}\right) \geq \frac{a_k}{2} n^k$

  (si usa il fatto che $n \geq n_0 \geq \frac{2k\gamma}{ak}$)

Ad esempio: $(n+1)^5 \in \Theta\left(n^5\right)$ ([GTG14,R-4.21])

Proprietà degli ordini di grandezza

Proposizione

$n^k \in o(a^n)$ se $k > 0, a > 1$ sono costanti.

Dimostrazione

Deriva immediatamente dal fatto che $\lim_{n\to+\infty} \frac{n^k}{a^n} = 0$

Proposizione

$(\log_b n)^k \in o\left(n^h\right)$ se $b > 1, k, h > 0$ sono costanti.

Dimostrazione

Ponendo $m = \log_b n$ si ha che $n = b^m$ e $n^h = (b^m)^h = \left(b^h\right)^m$. Sia $a = b^h > 1$. Dalla proprietà precedente deduciamo che

$$(\log_b n)^k = m^k \in o(a^m) = o\left(n^h\right).$$

Proprietà degli ordini di grandezza

Proposizione

Se $a, b > 1$ sono costanti allora $\log_b n \in \Theta(\log_a n)$

Dimostrazione

Deriva immediatamente dal fatto che $\log_b n = (\log_a n)(\log_b a)$.

Osservazione 1: Grazie alla proprietà, negli ordini di grandezza si omette la base dei logaritmi se si sa che essa è costante. Se invece la base del logaritmo viene omessa al di fuori di un ordine di grandezza, si assume base $b = 2$.

Osservazione 2: Si noti che per $a > 0$

$$a^{\log_2 n} = \left(2^{\log_2 a}\right)^{\log_2 n} = n^{\log_2 a}.$$

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 $x \bmod y$ = 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

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \text{(Serie di Gauss)}$$

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

$$\sum_{i=1}^{n} a^i = \frac{a^{n+1}-1}{a-1} - 1 = \frac{a^{n+1}-a}{a-1}$$

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

In ogni caso, 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 il più vicino possibile alla complessità vera (*tight bound*)
- $f(n)$ deve essere il più semplice possibile $\Rightarrow$ no costanti, no termini additivi di ordine inferiore. *Solo termini essenziali!*

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: $\Theta(n^k), k \geq 1$ costante
- esponenziale: $\Omega(a^n), a > 1$ costante

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

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

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

Nel prossimo lucido vedremo 2 algoritmi che risolvono in problema con complessità rispettivamente quadratica (algoritmo banale basato sulla definizione) e lineare (algoritmo efficiente che evita di ricalcolare più volte la stessa quantità). Per entrambi gli algoritmi vale la seguente specifica di input-output:

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

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

Algoritmo `prefixAverages1`

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

Complessità

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

$\Rightarrow$ quadratica

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

Complessità

$$\Theta(n)$$

$\Rightarrow$ lineare

Esercizio

Sia A un algoritmo che ricevuto in input un array X di n 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

Risoluzione

- $O\left(n^2\right)$: $\forall$ intero in X si eseguono $\leq c_1 n$ operazioni
  (N.B.: per n abbastanza grande, $c_1 n \geq c_2 \lceil \log_2 n \rceil$)
- $\Omega\left(n \log n\right)$: $\forall$ intero in X si eseguono $\geq c_2 \lceil \log_2 n \rceil$ operazioni
  (N.B.: vale per ogni istanza)
- $\Omega\left(n^2\right)$: X con tutti pari $\Rightarrow$ $\forall$ intero in X si eseguono $c_1 n$ operazioni
  (N.B.: vale per un'istanza specifica)

$\Rightarrow$ La complessità in tempo di A è $\Theta\left(n^2\right)$

Esercizio

Il seguente pseudocodice descrive l'algoritmo di ordinamento chiamato *InsertionSort*

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
}
```

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 sia

$$n_\tau = \text{max taglia eseguibile in tempo } \tau$$

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

Esempi

- $\log_2 n_\tau = \tau \Rightarrow n_\tau = 2^\tau$
- $n_\tau = \tau$
- $n_\tau^2 = \tau \Rightarrow n_\tau = \sqrt{\tau}$
- $2^{n_\tau} = \tau \Rightarrow n_\tau = \log_2 \tau$

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

Giustificazione $\approx$[GTG14,4.3.2]

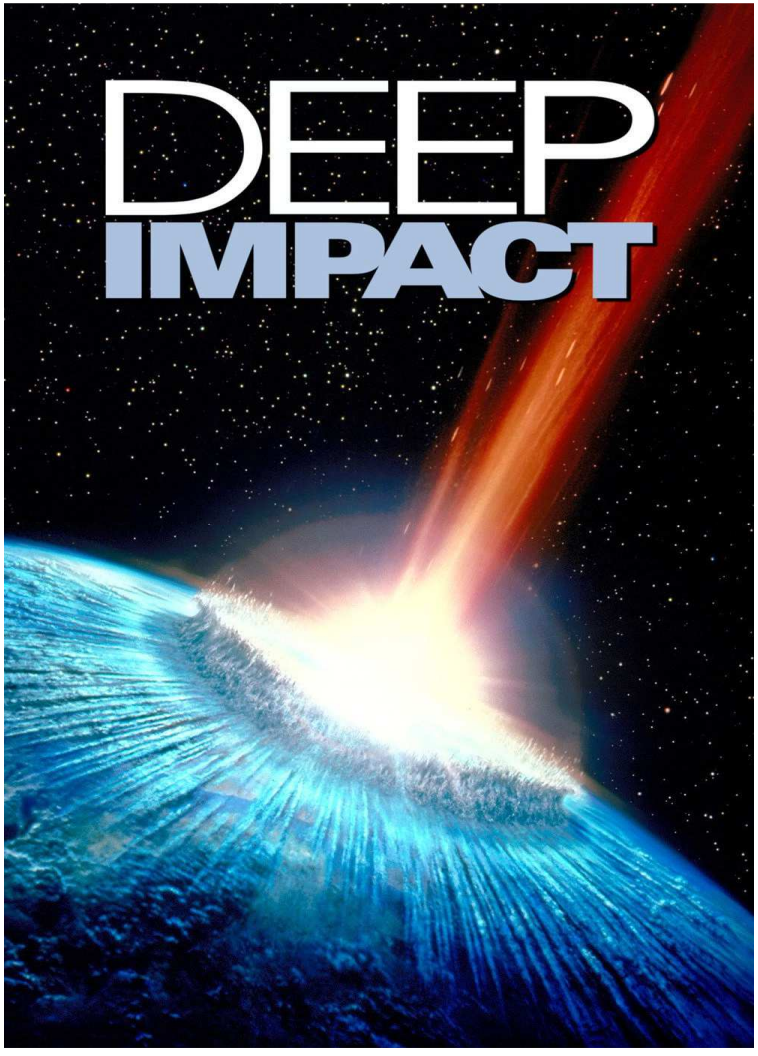
Supponiamo che la complessità $t_{\mathtt{A}}(n)$ sia espressa in ns, e sia

$$n_\tau = \text{max taglia eseguibile in tempo } \tau$$

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

$t_{\mathtt{A}}(n)$	$n_\tau : \tau = 10^9$ ns (1 sec)	$n_\tau : \tau = 60 \times 10^9$ ns (1 min)	$n_\tau : \tau = 3600 \times 10^9$ ns 1 hour
$\log_2 n$	$2^{10^9} = \infty$	∞	∞
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

N.B.: numero di atomi nell'universo $= 10^{78} \div 10^{82} = 2^{259} \div 2^{272}$

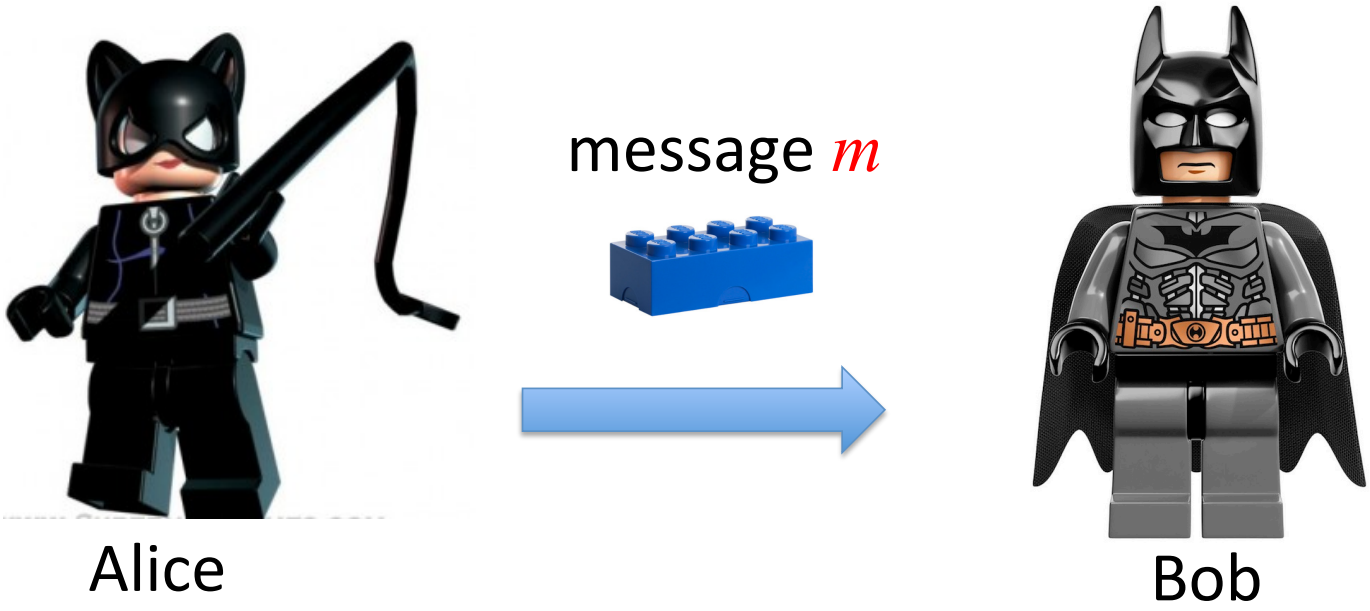


un algoritmo (o meglio la sua assenza) ha rivoluzionato la nostra società

un algoritmo potrebbe distruggerla

Esempio: sicurezza in internet

Crittografia a chiave pubblica



Bob: chiave privata k_1, 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

Algoritmo RSA (Rivest, Shamir, Adleman, 1977)

$N = p \cdot q$, dove p, q numeri primi grandi: $p, q \in \Theta\left(\sqrt{N}\right)$

Chiave pubblica: N, e ($e < N$ funzione di p, q)

Chiave privata: N, d ($d < N$ funzione di p, q)

messaggio m:

- cifratura: $m \to (m^e) \bmod N = \bar{m}$
- decifratura: $\bar{m} \to \left(\bar{m}^d\right) \bmod N = m$

N.B.: pur conoscendo N ed e, per conoscere d, necessario per la decifratura, mi serve conoscere p e q

Ottenere p e q da N è difficile!

Taglia dell'istanza = numero di bit di $N = \lfloor \log_2 N \rfloor + 1 \doteq n$

Algoritmo banale

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

Se $p, q \in \Theta\left(\sqrt{N}\right) \Rightarrow$ la complessità è $\Theta\left(\sqrt{N}\right) = \Theta\left(2^{\frac{n}{2}}\right)$.

Esempio numerico

- $n = 1024 \Rightarrow 2^{\frac{n}{2}} = 2^{512} \geq 10^{154}$
- Sul computer più potente al mondo ($\approx 10^{18}$ flop/sec) l'algoritmo banale richiederebbe circa $10^{154}/10^{18} = 10^{136}$ sec
- 1 anno ha $\leq 10^8$ sec $\Rightarrow > 10^{128}$ anni di calcolo ...

Osservazione

Esistono algoritmi più efficienti ma sempre con complessità esponenziale. 2009: risolto problema con $n = 768$ (2 anni di calcolo, centinaia di macchine). Non si può escludere l'esistenza di un algoritmo efficiente.

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

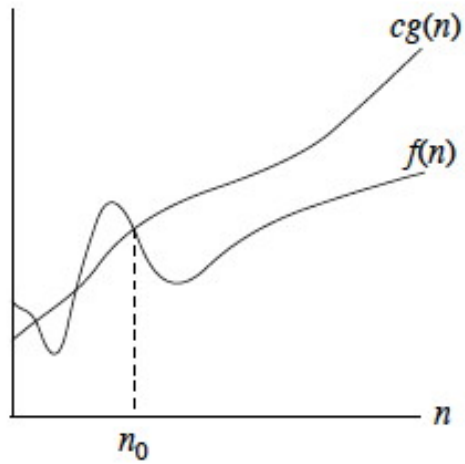
RSA is used in almost all internet-based commercial transactions. Without it, commercial online activities would not be as widespread as they are today. It allows users to communicate sensitive information like credit card numbers over an unsecure internet without having to agree on a shared secret key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

Analisi al caso pessimo (la più usata)

Potrebbe riguardare solo istanze *patologiche* mentre per tutte le istanze di interesse la complessità potrebbe migliorare (es: `Quicksort`)

Soluzioni

- restringere con opportune assunzioni il dominio delle istanze per mantenere quelle di interesse ed escludere quelle patologiche
- analisi del caso medio o cambiare probabilisticamente l'esecuzione

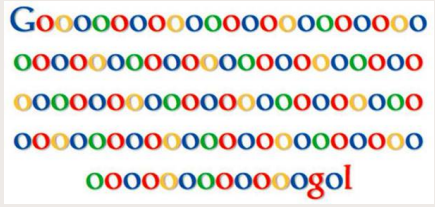
Caveat

Analisi asintotica (la più usata)

Le costanti trascurate potrebbero avere un impatto cruciale in pratica.

Esempio

- $t_{\mathtt{A}}(n) = 10^{100} n = \Theta(n)$ (10^{100} = "costante") $\qquad t_{\mathtt{B}}(n) = n^2$
  $\Rightarrow t_{\mathtt{A}}(n) = o(t_{\mathtt{B}}(n)) \Rightarrow$ `A` è più efficiente di `B`



- in realtà: $t_{\mathtt{A}}(n) < t_{\mathtt{B}}(n)$ solo se $n > 10^{100}$
  $10^{100} >>$ numero di atomi dell'universo...

Esempio

- $t_{\mathtt{A}}(n) = 400 \log_2 n \qquad t_{\mathtt{B}}(n) = \log_2^2 n$
  $t_{\mathtt{A}}(n) < t_{\mathtt{B}}(n) \Rightarrow \log_2 n > 400 \Rightarrow n > 2^{400} \geq 10^{100} \ldots$

Soluzioni: dare una stima delle costanti nel caso siano molto elevate

Tecniche di dimostrazione [GTG14, Par. 4.4]

Esempio/Controesempio/Assurdo

Nelle analisi di complessità e correttezza, si usano spesso argomenti basati su esempi o controesempi e argomenti per assurdo. Illustriamo queste tre tecniche.

- (Dimostrazione tramite esempio) Per dimostrare che la complessità di un algoritmo è $\Omega(f(n))$ basta far vedere che, per ogni n abbastanza grande, esiste un'istanza che richiede $\geq cf(n)$ operazioni
- (Dimostrazione tramite controesempio) Per confutare l'affermazione "$2^i - 1$ è primo $\forall i \geq 1$" è sufficiente osservare che $2^4 - 1 = 15$
- (Dimostrazione per assurdo) Per dimostrare che, dati due interi $a, b \geq 1$, se ab dispari allora sia a che b sono dispari procediamo come segue: se almeno uno tra a o b fosse pari (negazione della tesi) , ad es. a, allora $a = 2c$ per un qualche intero $c \geq 1$, e quindi $ab = 2cb$ che è pari, contraddicendo l'ipotesi.

Induzione

Per provare che una proprietà $Q(n)$ è vera $\forall n \geq n_0$ ($n_0 = 1$ in [GTG14]) Si procede così:

- scegli opportunamente $k \geq 0$
- *base*: dimostra $Q(n_0), Q(n_0 + 1), \ldots, Q(n_0 + k)$
- *passo induttivo*: si fissa $n \geq n_0 + k$ *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) $k = 0$

Esempio (Induzione)

$Q(n)$: "$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \in \Theta\left(n^2\right)$", $\forall n \geq 0$ ($\Rightarrow n_0 = 0$)

- $k = 0$
- *base*: $Q(0) = \sum_{i=0}^{0} i = 0 = \frac{0 \cdot 1}{2} \Rightarrow$ OK
- *passo induttivo*: fissa $n \geq 0$
  Ipotesi induttiva: $Q(m)$ vera $\forall m : 0 \leq m \leq n$
  $Q(n+1)$:
  $\sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + (n+1) = (n+1) + \frac{n(n+1)}{2} = \frac{(n+1)(n+2)}{2}$
  - penultimo "=": per ipotesi induttiva

Esercizio [GTG14, C-4.35]

Dimostrare la seguente proprietà:
$Q(n)$: "$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \in \Theta\left(n^3\right)$", $\forall n \geq 0$

Esempio: successione di Fibonacci

Successione di Fibonacci:

$F(0) = 0 \qquad F(1) = 1$

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

(In [GTG14] Prop. 4.20: definita in modo non standard)

$F(n)$ = numero coppie di conigli all'inizio del mese n ($n \geq 1$)

- una coppia produce un'altra coppia ogni mese
- una coppia diventa fertile dopo due mesi di vita
- i conigli non muoiono
- all'inizio del primo mese esiste una coppia neonata

Claim

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

$\Phi = \frac{1+\sqrt{5}}{2} \qquad \hat{\Phi} = \frac{1-\sqrt{5}}{2} \qquad (\Phi = \textit{golden ratio})$

Dimostrazione

$n_0 = 0, k = 1$

Base: $n = 0, n = 1$ (esercizio)

Passo induttivo:

Hp. induttiva: $F(m) = \frac{1}{\sqrt{5}}\left(\Phi^m - \hat{\Phi}^m\right)$ per $0 \leq m \leq n$,

con $n \geq 1 = n_0 + k$.

$F(n+1)$?

$$F(n+1) \overset{(\text{def})}{=} F(n) + F(n-1)$$
$$\overset{(\text{hp. ind})}{=} \frac{1}{\sqrt{5}}\left(\Phi^n + \Phi^{n-1} - \left(\hat{\Phi}^n + \hat{\Phi}^{n-1}\right)\right)$$

$$\Phi^n + \Phi^{n-1} = \Phi^{n-1}(\Phi + 1) = \Phi^{n-1}\left(\frac{1+\sqrt{5}}{2} + 1\right) = \Phi^{n-1}\frac{3+\sqrt{5}}{2}$$

$$\frac{3+\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^2 \Rightarrow \Phi^n + \Phi^{n-1} = \Phi^{n-1} \cdot \Phi^2 = \Phi^{n+1}$$

Analogamente: $\hat{\Phi}^n + \hat{\Phi}^{n-1} = \hat{\Phi}^{n+1}$

$$\Rightarrow F(n+1) = \frac{1}{\sqrt{5}}\left(\Phi^{n+1} - \hat{\Phi}^{n+1}\right)$$

□

Induzione fallace

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

- $k = 0$
- *base*: $F(0) = 0 \Rightarrow$ OK
- *induzione*: Fissiamo $n \geq 0$
  Hp. induttiva: $F(m) = 0, \forall m : 0 \leq m \leq n$
  $F(n+1) = F(n) + F(n-1) = 0 + 0 = 0$

Dov'è l'errore?

$F(n+1) = F(n) + F(n-1)$ non vale $\forall n \geq 0$ ma vale $\forall n \geq 1$

Correttezza

Approccio generale all'analisi (di correttezza) di un algoritmo

- Identificare 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)

Osservazione

Per provare la terminazione è sufficiente assicurarsi che i cicli (inclusi GOTO) abbiano termine.

Invarianti [GTG14, Par. 4.4.3]

Definizione

Un *invariante* per un ciclo è una proprietà espressa in funzione delle variabili usate nel ciclo, che deve valere all'inizio del ciclo e alla fine di ciascuna iterazione e che, dopo l'ultima iterazione, garantisce la correttezza del ciclo.

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

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

Proprietá: $x \notin \{A[0], A[1], \ldots, A[i-1]\}$

Dato l'invariante si procede così:

- si dimostra che è vero all'inizio del ciclo
- si dimostra che se vale prima dell'inizio di una arbitraria iterazione (cioè alla fine della precedente) allora esso vale anche alla fine dell'iterazione
- si dimostra che se vale alla fine dell'ultima iterazione, allora il ciclo è corretto

Esempio: `ArrayMax(A)`

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

Invariante per il ciclo for

currMax $= \max\{A[j] : 0 \leq j \leq i\}$

- All'inizio del ciclo ($i = 0$) è vero per l'inizializzazione di currMax
- Se è vero alla fine della iterazione $i-1$ allora a currMax verrà assegnato il valore massimo tra $A[i]$ e il massimo in $A[0], A[1], \ldots, A[i-1]$, e quindi l'invariante rimane vero.
- Se è vero alla fine del ciclo ($i = n-1$) allora currMax è il massimo valore nell'array, che è la proprietà finale di correttezza del ciclo.

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

Invariante per il ciclo for

$x \notin \{A[0], A[1], \ldots, A[i-1]\}$

La proprietà finale di correttezza da provare è: il valore restituito è corretto.

Dimostrazione della correttezza tramite l'invariante

- All'inizio ($i = 0$) l'invariante è vero per vacuità (l'insieme $\{A[0], A[1], \ldots, A[i-1]\}$ è vuoto)
- Supponiamo che l'invariante valga alla fine di una iterazione e consideriamo la successiva iterazione (con valore i della variabile di ciclo). Se in tale iterazione si scopre che $x = A[i]$ allora l'iterazione è l'ultima eseguita e si restituisce correttamente il valore i. Se invece $x \neq A[i]$ allora i viene incrementato di 1. In ogni caso l'invariante continua a valere.
- Sia k ultimo valore assunto dalla variabile i: Se $k < n$ significa che si è usciti dal while trovando $x = A[k]$, altrimenti $k = n$ e l'invariante garantisce che $x \notin A$. In entrambi i casi, il valore restituito in output è corretto.

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.

Svolgimento

La proprietà finale di correttezza è: max contiene la lunghezza del più lungo segmento continuo di 1 in $S[1 \ldots n]$

Invariante:

- max contiene la lunghezza del più lungo segmento continuo di 1 in $S[1 \ldots i]$
- curr contiene la lunghezza del più lungo suffisso continuo di 1 in $S[1 \ldots i]$.

Dimostrazione:

- All'inizio ($i = 0$) l'invariante è vero *per vacuità* ($S[1 \ldots i]$ è vuoto).
- Supponiamo che l'invariante sia vero alla fine della iterazione $i - 1$ e consideriamo l'iterazione i. Verificare per esercizio che le operazioni fatte sia nel caso $S[i] = 1$ che nel caso $S[i] = 0$ assicurano che l'invariante rimanga vero alla fine del ciclo.
- Se l'invariante è vero alla fine del ciclo, la prima proprietà dell'invariante implica la proprietà finale desiderata.

Esercizi

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 vale la proprietà che in ciascuna riga gli 1 vengono prima degli 0. Si supponga anche che il numero di 1 nella riga i sia maggiore o uguale al numero di 1 nella riga $i+1$, per $i = 0, 1, \ldots, n-2$. 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 [GTG14, Par. 5.1-5.4]

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*)
- ricorrendo a soluzione di ≥ 1 istanze di taglia $< n$: se $n > n_0 + k$

Ad esempio MergeSort (che riprenderemo più avanti) ordina una sequenza S direttamente quando contiene un solo elemento ($n_0 = 1, k = 0$), oppure ordinando separatamente le due metà (di taglia $\leq \lceil n/2 \rceil < n$) e poi fondendole una volta ordinate.

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

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

Esecuzione di un algoritmo ricorsivo

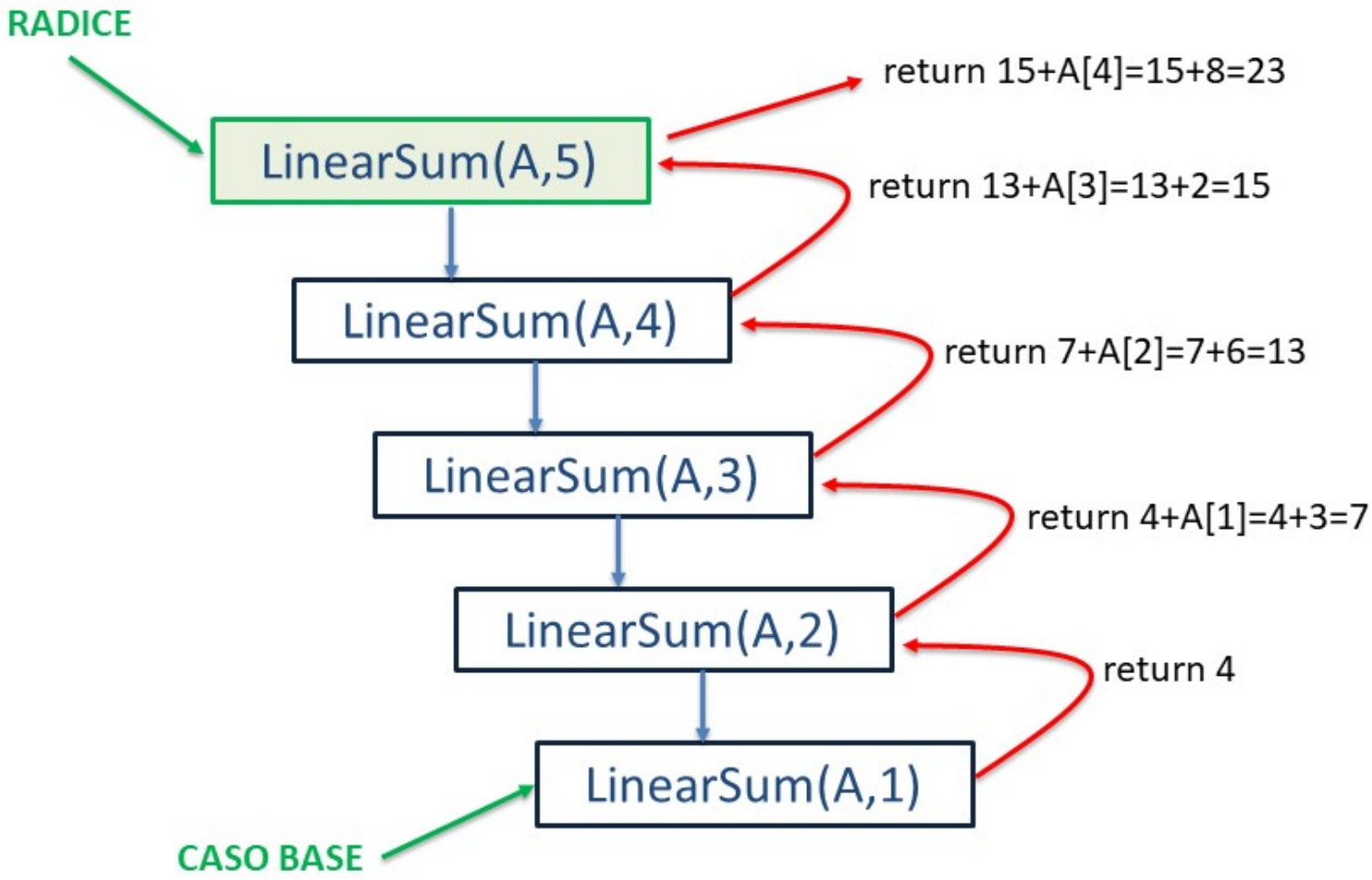
Albero della ricorsione

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

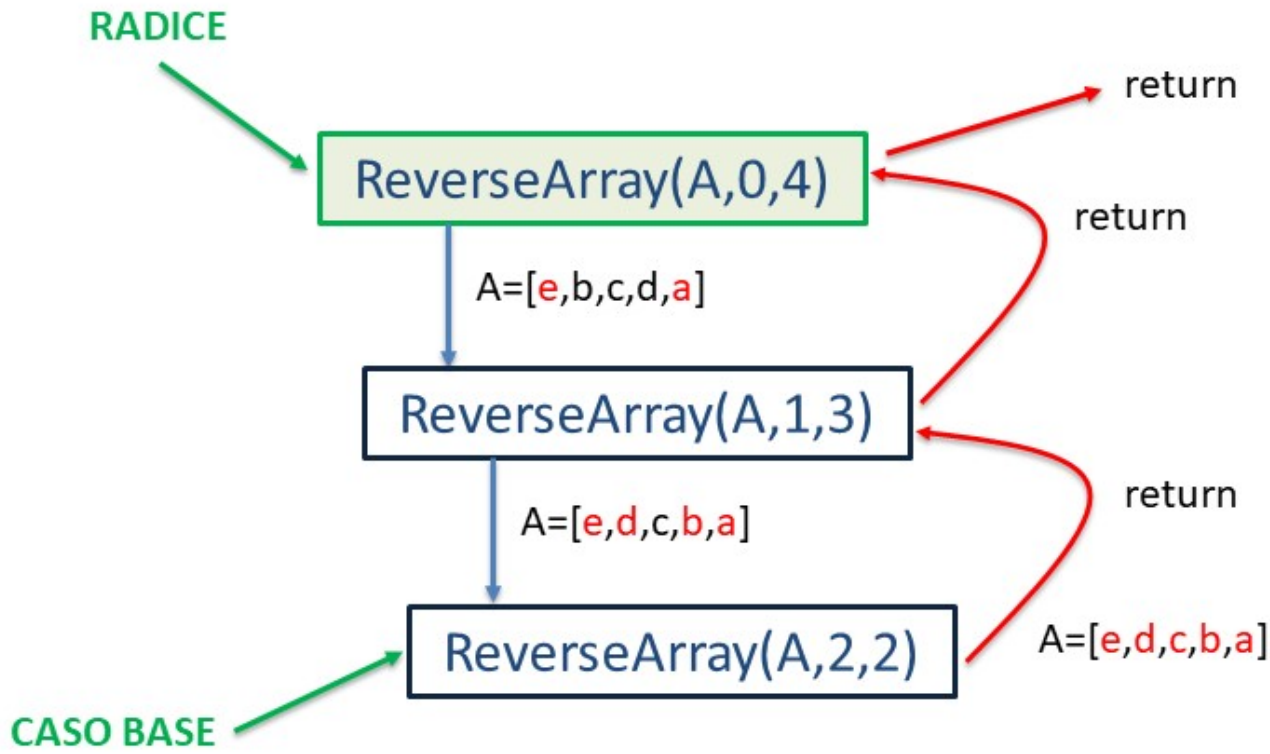
- Ogni nodo corrisponde a un'invocazione ricorsiva distinta fatta durante l'esecuzione dell'algoritmo.
- La radice dell'albero corrisponde alla prima invocazione, e i figli di un nodo x sono associati alle invocazioni ricorsive fatte direttamente dall'invocazione corrispondente a x.
- Le foglie dell'albero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

N.B. [GTG14] usa il termine *recursion trace* per denotare l'albero della ricorsione.

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]

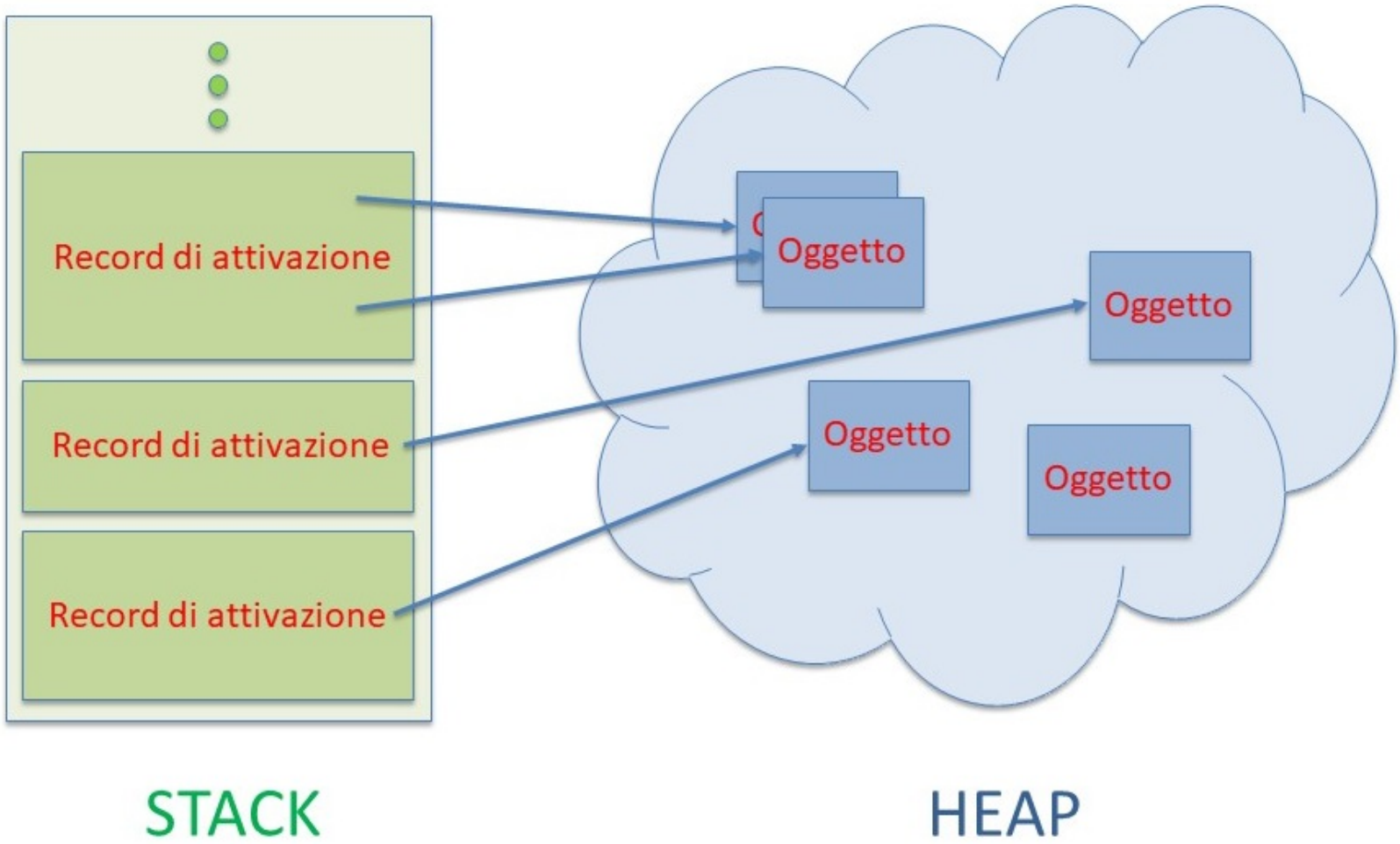


Esecuzione di un algoritmo ricorsivo

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

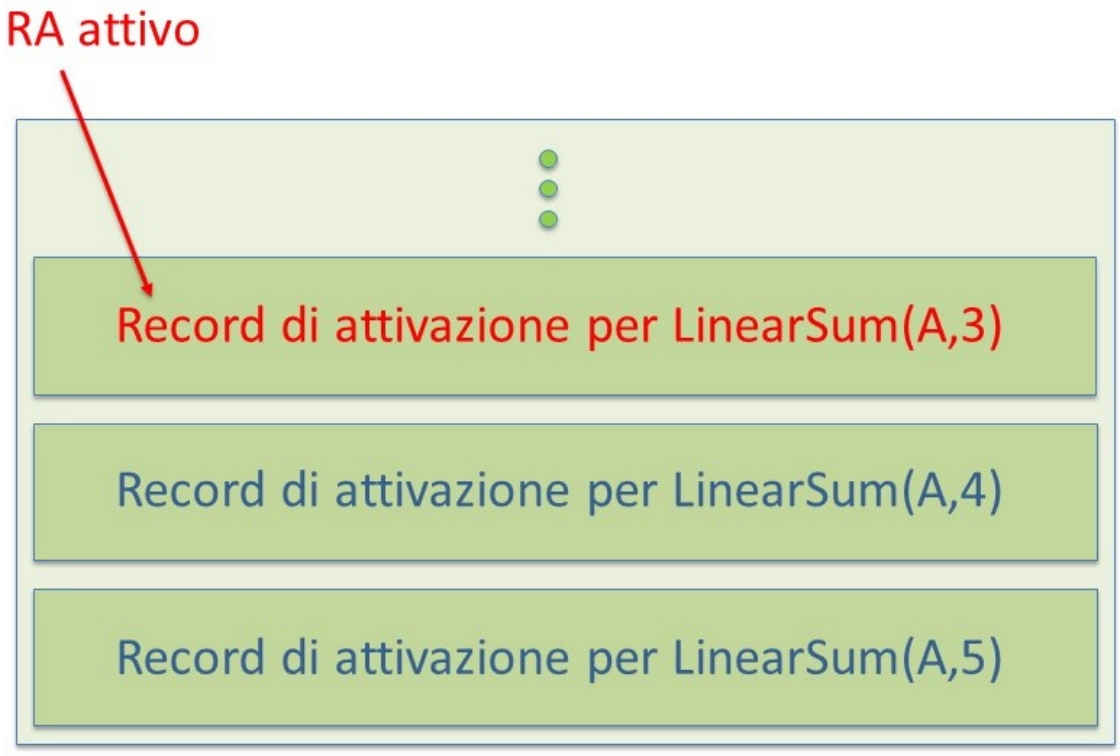
- STACK: spazio destinato a memorizzare variabili locali ai metodi e riferimenti a oggetti.
  - Per ogni invocazione di un metodo viene inserito un record di attivazione (RA) contenente variabili e riferimenti a oggetti relativi a quella invocazione.
  - Un RA viene eliminato quando l'invocazione di metodo corrispondete finisce l'esecuzione.
  - I RA sono inseriti/eliminati con politica LIFO, e in ogni istante possono essere acceduti solo i dati realtivi all'ultimo RA inserito.
- 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:



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 massimizza su tutte le possibili istanze i di taglia n (ovvero, basta fare una stima superiore che vale per tutte le istanze di taglia n), mentre per ottenere un *lower bound* a $t_A(n)$ si identifica un'istanza particolare o si fa una stima inferiore che va bene per tutte le istanze.

Esempio

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

Complessità:

- Per ogni istanza di taglia n l'albero della ricorsione ha n nodi
- Ogni invocazione di `LinearSum` esegue $\Theta(1)$ operazioni, esclusa l'eventuale chiamata ricorsiva. Quindi il costo associato a ciascun nodo è $\Theta(1)$.

$\Rightarrow t_{\texttt{LinearSum}}(n) \in \Theta(n)$

Esercizio

Analizzare la complessità di `ReverseArray` utilizzando come taglia dell'istanza la lunghezza del segmento da ribaltare (la taglia dell'istanza per l'invocazione `ReverseArray`(A, i, j) con $j \geq i$ è $n = j - i + 1$.

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;
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 $n > 1$ (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 $\Theta(1)$.

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

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

Dimostriamo la correttezza di `LinearSum`(A, n) per ogni $n \geq 1$.

- base ($n = 1$): OK
- passo induttivo: fissiamo $n \geq 1$ e assumiamo (hp. induttiva) che `LinearSum`(A, m) sia corretto per ogni $1 \leq m \leq n$.

  Consideriamo l'esecuzione di `LinearSum`$(A, n+1)$. Grazie alla hp. induttiva `LinearSum`(A, n) calcola correttamente la somma dei primi n valori e aggiungendo poi $A[n]$ si ottiene il valore corretto.

Esercizio

Provare la correttezza di `ReverseArray` e `Power`.

Complessità di algoritmi ricorsivi (2)

Esistono altri metodi per analizzare la complessità di algoritmi ricorsivi. Ad esempio:

- Ipotizzare un upper o lower bound alla complessità (guess), e provarlo per induzione.
- Descrivere la complessità tramite una relazione di ricorrenza e usare una delle tecniche note per la "risoluzione" di relazioni di ricorrenza. Questo metodo sarà studiato nel corso di *Algoritmi per l'Ingegneria*.

Come esempio del primo metodo, consideriamo il seguente algoritmo ricorsivo per i numeri di Fibonacci.

Algoritmo `RecurFib(`n`)`
Input: intero $n \geq 0$
Output: $F(n)$
if $n \leq 1$ **then return** n;
return `RecurFib(`$n-1$`)+RecurFib(`$n-2$`)`

Analisi di RecurFib

Sia $t_{\texttt{RecurFib}}(n)$ la complessità di `RecurFib(`n`)` (in questo caso, rappresenta il numero di operazioni eseguite dall'algoritmo per risolvere l'unica istanza di taglia n).

Dimostriamo per induzione su $n \geq 0$, che $t_{\texttt{RecurFib}}(n) \geq F(n)$.

- *Base*: Per $n = 0, 1$ il lower bound vale in base all'ipotesi fatta.
- *Passo induttivo*: fissiamo $n \geq 1$, e assumiamo come ipotesi induttiva che $t_{\texttt{RecurFib}}(m) \geq F(m)$ per ogni $0 \leq m \leq n$. Consideriamo adesso $t_{\texttt{RecurFib}}(n+1)$. Poiché `RecurFib(`$n+1$`)` chiama al suo interno `RecurFib(`n`)` e `RecurFib(`$n-1$`)`, allora è evidente che

$$\begin{aligned} t_{\texttt{RecurFib}}(n+1) &\geq t_{\texttt{RecurFib}}(n) + t_{\texttt{RecurFib}}(n-1) \\ &\geq F(n) + F(n-1) \quad \text{(per hp. induttiva)} \\ &= F(n+1). \end{aligned}$$

Esercizio

Disegnare l'albero della ricorsione per $t_{\texttt{RecurFib}}(6)$.

Osservazioni

`RecurFib` è inefficiente perché ricalcola molte volte lo stesso valore (come si vede dall'albero della ricorsione).

Il seguente algoritmo, che risulta molto più 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: $F(n)$
$\Phi \leftarrow \big((1+\sqrt{5})/2\big)$;
$\hat{\Phi} \leftarrow \big((1-\sqrt{5})/2\big)$;
return $(\texttt{Power}(\Phi, n) - \texttt{Power}(\hat{\Phi}, n))/\sqrt{5}$

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

Esercizi

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` eficiente 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

Esempio domande prima parte

- Si definisca il problema di trovare il massimo in una sequenza di interi come problema computazionale.
- Sia `A` un algoritmo per un problema computazionale Π. Si supponga che per ogni n esista un'istanza di taglia n che richiede $\geq n^2$ operazioni. Cosa si può dire della complessità al caso pessimo di `A`?
- Siano `A` e `B` due algoritmi che risolvono lo stesso problema computazionale Π, dove la complessità al caso pessimo di `A` è $O(n)$ e quella di `B` è $\Omega(n^2)$. Per ogni $n \geq 1$, sia denoti con i_n l'istanza *peggiore* per `B`, e siano $t_{\mathtt{A},i_n}$ e $t_{\mathtt{B},i_n}$ il tempo di esecuzione, rispettivamente di `A` e di `B`, sull'istanza i_n. Si può affermare che per n abbastanza grande $t_{\mathtt{B},i_n} \geq t_{\mathtt{A},i_n}$?
- Definire la nozione di albero della ricorsione per gli algoritmi ricorsivi.

Errata

Cambiamenti rispetto alla prima versione dei lucidi:

- Lucido 17: nella definizione di $\mathcal{S}$ "$i, j \in Z$" diventa "$j, k \in Z$ con $j \geq -1$ e $k \geq 0$"
- Lucido 19: nella definizione di $\mathcal{S}$ è stata tolta la condizione $|Y| = k$.
- Lucido 25: eliminata la seconda osservazione.
- Lucido 46: aggiunta l'ipotesi che c_1 e c_2 sono costanti postive.
- Lucidi 66, 70, 73, e 74: modificato leggermente il testo.
- Lucido 72: aggiunto nome dell'algoritmo e specifica input/output
- Lucido 76: (nella riga 5) n_0 diventa $n_0 + k$.
- Lucido 89: $A[n+1]$ diventa $A[n]$.