Algoritmi e Complessità

A.A. 2017-2018

Bibliografia

- J. Kleinberg, E. Tardos, “Algorithm Design”, Pearson Education, 2014, Cap. 6,
ISBN 13: 978-1-292-02394-6.

- T. H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, "Introduzione agli algoritmi e strutture dati", Seconda edizione, McGraw-Hill, 2005, Cap. 15 e Cap. 25.

- A. Bertossi, A. Montresor "Algoritmi e strutture di dati", Citta` studi Edizioni, 2010, Cap. 13.

- E. Horowitz, S. Sahni, “Fundamentals of Computer Algorithms”, Computer Science Press, 1978, Cap. 7 e Cap. 8.

- J. E. Hopcroft, R. Motowani, J. D. Ullman, “Automi, linguaggi e calcolabilità”, Addison-Wesley, 2003, ISBN 88-7192-154-2, Cap. 10.

Problemi e algoritmi

Soluzione di un problema computazionale:
- procedimento automatico (algoritmo)
- calcolare risultati (output) a partire dai dati del problema (input)
- utilizzare una quantità limitata di risorse

- nozione formale di procedimento automatico

  Macchina di Turing

  ***Random Access Machine* (RAM)**

- rappresentazioni di dati e risultati

- valutare il costo del procedimento

Linguaggio di disegno

- assegnazione: $\leftarrow$ ($i \leftarrow j \leftarrow k$ equivale alla seq.: $j \leftarrow k$; $i \leftarrow j$)
- espressioni numeriche e booleane
- dichiarazione e chiamata di metodo: nome (*param 1*, *param 2*, ...)
- ritorno da un metodo: return *valore*
- dati composti:
  - i-esimo elemento array A: A[i]
  - A[i . . j] $\equiv$ <A[i], A[i+1], . . . , A[j]> se $i \leq j$
    sequenza vuota se $i > j$
  - i dati composti sono organizzati in oggetti, che sono strutturati in attributi o campi: ad es. *length*[A]
- una variabile che rappresenta un oggetto è un riferimento
- un riferimento che non si riferisce a nessun oggetto: *nil*
- parametri alle procedure passati per valore (per gli oggetti una copia del riferimento)

Linguaggio di disegno

struttura di blocco: spaziatura
costrutti iterativi e condizionali:

if (condizione) azioni
[else if (condizione) azioni]
…
[else azioni]

while (condizione)
    azioni

for variabile ← val-iniz. to val-fin [incremento]
    azioni

for variabile ← val-iniz. downto val-fin [decremento]
    azioni

Algoritmi: complessità

Confronto tra algoritmi che risolvono un problema

- complessità o efficienza computazionale:

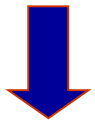
quantità di risorse usate durante la loro esecuzione

valutazione del tempo d'esecuzione indipendente dalla tecnologia dell'esecutore.

confronto di funzioni complessità computazionale asintotica

Algoritmi: complessità

Efficienza asintotica degli algoritmi: come cresce il tempo di esecuzione con il crescere *al limite* della dimensione delle istanze in input

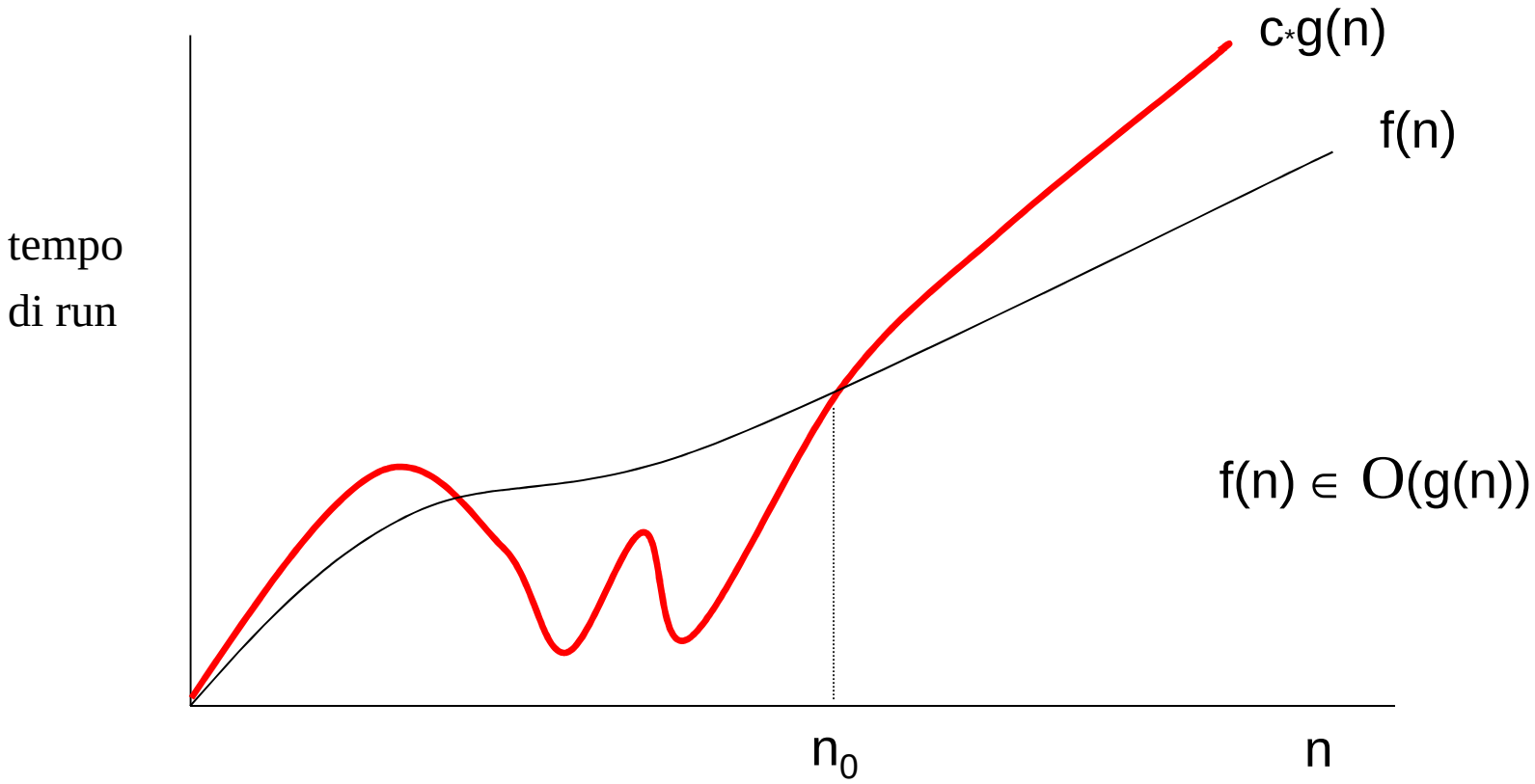


Notazione asintotica

Consideriamo funzioni dai naturali ai numeri reali non negativi

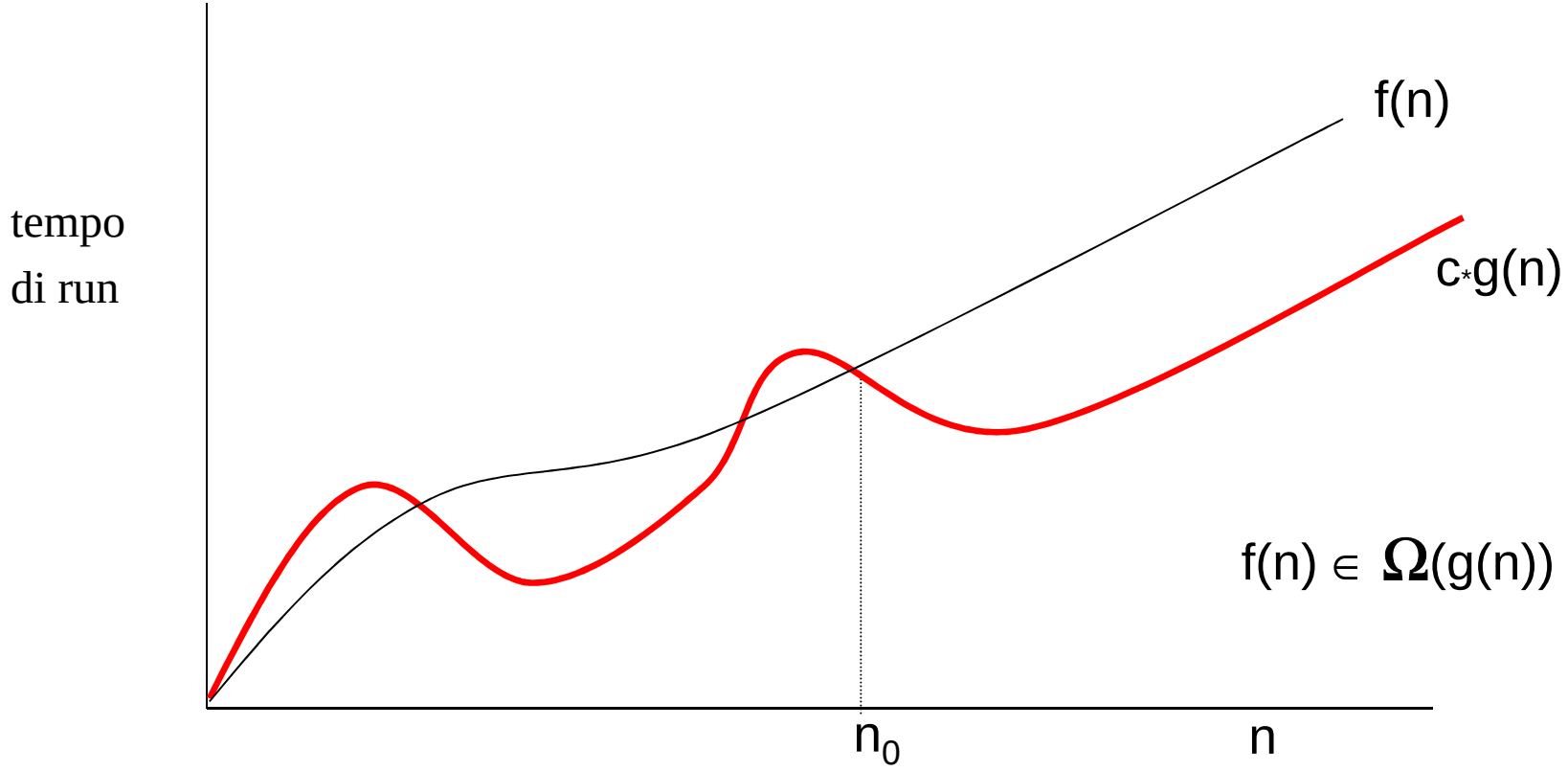
Notazione O: $O(g(n))$ è l'insieme di tutte le funzioni f(n) per cui esistono due costanti positive c ed n_0 tali che

$$f(n) \leq c * g(n) \quad \text{per tutti gli } n \geq n_0$$



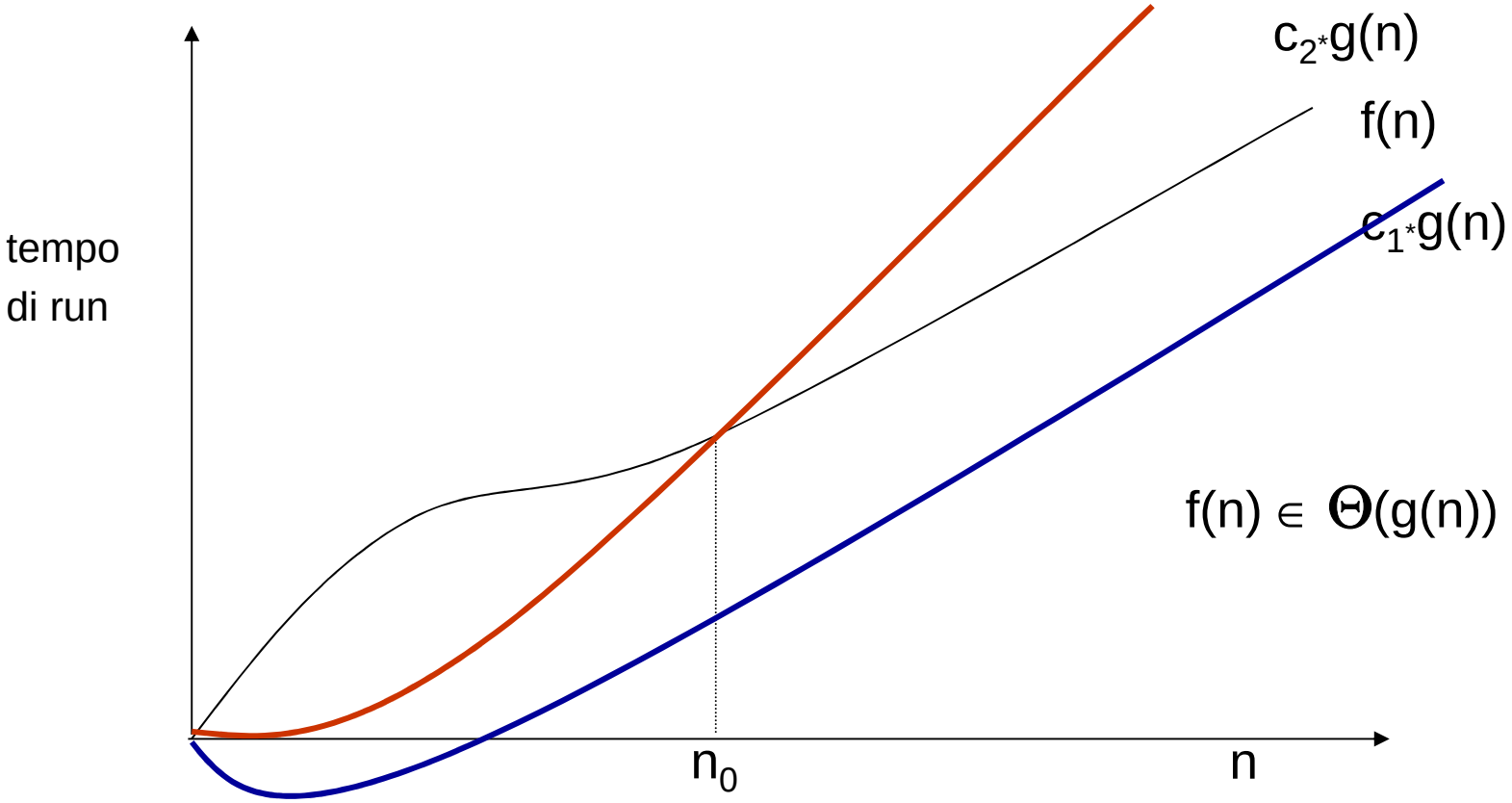
Notazione Ω: Ω(g(n)) è l'insieme di tutte le funzioni f(n) per cui esistono due costanti positive c ed n_0 tali che

$$f(n) \geq c * g(n) \quad \text{per tutti gli } n \geq n_0$$



Notazione Θ: Θ(g(n)) è l'insieme di tutte le funzioni f(n) per cui esistono tre costanti positive c_1, c_2 ed n_0 tali che

$$c_{1^*}\, g(n) \le f(n) \le c_{2\,*}\, g(n) \quad \text{per tutti gli } n \ge n_0$$



La tirannia del tasso di crescita

Tre algoritmi per risolvere un problema: $\boldsymbol{A}_1$, $\boldsymbol{A}_2$ e $\boldsymbol{A}_3$.

$$\mathbf{T}_1(n) \in \Theta(n) \quad \mathbf{T}_2(n) \in \Theta(n^5) \quad \mathbf{T}_3(n) \in \Theta(2^n).$$

Supponiamo che sia: $\mathbf{T_1}(n) = n$, $\mathbf{T_2}(n) = n^5$, $\mathbf{T_3}(n) = 2^n$, e che il tempo effettivo per eseguire l'algoritmo su un input di dimensione "1" sia 1μs (10^{-6} sec.).

	costo	input di dim. = 10	input di dim. = 60
$\boldsymbol{A_1}$	n	$= 10^{-5}$ sec.	$= 6 \times 10^{-5}$ sec.
$\boldsymbol{A_2}$	n^5	$= 0.1$ sec.	$\approx$ 13 min.
$\boldsymbol{A_3}$	2^n	$\approx 10^{-3}$ sec.	$\approx$ 366 secoli

Complessita` ed evoluzione tecnologica

L'algoritmo può essere **il fattore dominante** per le prestazioni di un programma e pesa può pesare molto più, ad esempio, della velocità della macchina.

	costo	vecchio computer	nuovo computer 1.000 volte più veloce
A_1	n	N_1	$N_1' = 1000\ N_1$
A_2	n^5	N_2	$N_2' \approx 4\ N_2$
A_3	2^n	N_3	$N_3' \approx N_3 + 10$

TEMPI DI ESECUZIONE IN SECONDI

Ampiezza dati input	$\log_2 n$	n	n^2	2^n
10	0.000003	0.000010	0.0001	0.001
100	0.000007	0.0001	0.01	10^{14} SECOLI ! !
1000	0.000010	0.001	1	incommen-surabile
10000	0.000013	0.01	102	incommen-surabile
100000	0.000017	0.1	1080	incommen-surabile

AMPIEZZA DELL'INPUT RISOLVIBILE IN UN DATO TEMPO

COMPLES-SITA'	COMPUTER ATTUALE	100 VOLTE + VELOCE	1000 VOLTE + VELOCE
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Introduzione

Normalmente si richiede che un algoritmo soddisfi tre condizioni:

(1) calcoli esattamente i risultati voluti
(2) utilizzi una quantità di risorse limitata
(3) faccia questo per ogni "possibile input .

Vi sono però situazioni in cui non è possibile soddisfare tutte e tre queste condizioni: in particolare non è possibile calcolare esattamente i risultati voluti quando dati e/o risultati sono oggetti che non ammettono una rappresentazione finita.

Ad esempio i numeri reali non si possono rappresentare in modo finito e quindi non si possono calcolare in modo esatto il valore di $\sqrt{2}$ o di π. Si deve lavorare con valori approssimati finitamente rappresentabili (ad esempio in virgola mobile).

Lo studio di algoritmi di approssimazione sui numeri reali è talmente importante da costituire un'intera disciplina: il Calcolo Numerico.

Aspetti di ottimizzazione

Ci interessano invece problemi computazionali in cui sia i dati in ingresso che i risultati sono finitamente rappresentabili.
Problemi di questo tipo sono molto comuni nelle applicazioni informatiche:

- ”Trovare le pagine web che sono maggiormente correlate ad una particolare parola chiave”, è alla base di ogni motore di ricerca;
- ”Trovare la strada più breve tra due punti di una mappa”, è alla base di ogni navigatore satellitare;
- ”Trovare il miglior modo per inviare un pacchetto da un nodo ad un altro di una rete”, è alla base di ogni router di rete; ecc.

Sono esempi di problemi di ottimizzazione discreta in cui non solo si vuole trovare una soluzione che soddisfi tutte le condizioni poste dal problema, ma tra di esse se ne vuole trovare una ottima rispetto ad un qualche parametro di ottimalità.

Introduzione

Purtroppo la maggior parte di questi problemi è **NP**-difficile e quindi ogni algoritmo che li risolve esattamente richiede (se **P≠NP**) un tempo di esecuzione almeno esponenziale nella dimensione.

Anche la soluzione di problemi che si risolvono esattamente con algoritmi di complessità polinomiale può risultare troppo costosa (un algoritmo di complessità (n^{10}) non può certo dirsi poco costoso).

Non è sempre possibile soddisfare contemporaneamente tutte e tre le condizioni e in questi casi almeno una delle tre deve essere rilasciata.

-Talvolta viene rilasciata la condizione (3): l'algoritmo risolve esattamente il problema computazionale in un tempo ragionevole soltanto per una particolare classe di dati in input.
La soluzione è soddisfacente soltanto se:

- è possibile specificare esattamente la classe di input su cui l'algoritmo funziona
- si è sicuri che l'algoritmo verrà usato soltanto con input appartenenti a tale classe

Un approccio più comune è rilasciare la condizione (2): accettare che per alcuni input l'algoritmo possa richiedere una quantità eccessiva di risorse. Per controllare che l'algoritmo funzioni bene nelle applicazioni pratiche di solito viene eseguito su di un insieme opportunamente scelto di input tipici, verificando che su questi input esso fornisca il risultato in un tempo ragionevole. Se funziona bene sugli input tipici si può sperare che esso funzioni bene con tutti gli input che si presenteranno nella pratica.
Questo è l'approccio generalmente usato per risolvere problemi complessi di ricerca operativa, intelligenza artificiale, programmazione con vincoli, ecc. Il problema con questo approccio è che non si ha la garanzia che l'algoritmo funzioni bene nelle applicazioni pratiche.

Un terzo approccio è quello di rilasciare la condizione (1): accettare che per ogni possibile input l'algoritmo calcoli con risorse limitate dei risultati approssimati che non si discostino troppo dai risultati esatti o che in alcuni casi i risultati siano errati.

Algoritmi: obiettivi del corso

1. Tecniche algoritmiche

- ***Greedy***

  Approccio “ingordo”: si fa sempre la scelta localmente ottima.

- ***Divide-et-impera***

  Un problema viene suddiviso in sottoproblemi *indipendenti* che vengono risolti ricorsivamente (top-down) (problemi di *decisione*, di *ricerca*).

- ***Backtracking***

  Si procede per “tentativi”, tornando ogni tanto sui passi precedenti.

- ***Programmazione dinamica***

  La soluzione viene costruita a partire da un insieme di sotto problemi *potenzialmente ripetuti* (bottom up) (problemi di *ottimizzazione*).

- ***Ricerca locale***

  La soluzione ottima viene trovata “migliorando” soluzioni esistenti non ottime.

2. Soluzioni “ragionevoli” per problemi “difficili”

Algoritmi di approssimazione: algoritmi che calcolano risultati approssimati che non si discostano troppo dai risultati esatti.

Algoritmi randomizzati: algoritmi che calcolano risultati esatti con una certa probabilità.

Simulated annealing, ***ricerca tabù***, ***algoritmi genetici***: usati per risolvere problemi di ottimizzazione, specialmente quelli combinatori.

Consistono di principi di ricerca organizzati in una strategia generale e pertanto non possono essere descritti come algoritmi, ma piuttosto come metodi o meta-algoritmi.

Spesso vengono chiamati con i termini “metaeuristiche” o “euristiche generali”. I nomi e i principi di queste euristiche sono ispirati da processi o concetti che non hanno niente a che fare con la programmazione.