Analisi di algoritmi

(Si può fare meglio di così?)

Emanuele Nardi

Compilato il 29 giugno 2019
v1.0.0

1 Introduzione

Il nostro obiettivo è stimare la complessità *in tempo* degli algoritmi. Dovremmo stimare anche quella in spazio, ma la complessità in spazio dipende da quella in tempo. Daremo delle definizioni, parleremo di modelli di calcolo, faremo qualche esempio di valutazione precisa e introdurremo una notazione.
Faremo tutto questo per stimare il tempo per un dato input, per stimare il più grande input gestibile in tempi ragionevoli, per avere un meodo di misura per confrontare algoritmi diversi e in particolare per ottimizzare le parti più importanti dell'algoritmo.

1.1 Definizione di complessità

La complessità viene definita come una funzione che data la dimensione dell'input restituisce il tempo considerato come un valore intero.

> Come definiamo la dimensione dell'input? Come misuriamo il tempo?

1.2 Valutare la dimensione dell'input

Esistono due criteri per valutare la dimensione dell'input:

1. Il criterio di costo logaritmico: dove la dimensione dell'input è il numero di bit necessari per rappresentarlo (un esempio è la moltiplicazione di numeri binari lunghi n bit);
2. Il criterio di costo uniforme: dove la dimensione dell'input è il numero di elementi di cui è costituito (un esempio è a ricerca del minimo in un vettote di n elementi).

Ad esempio consideriamo n interi rappresentati tramite 32 bit. Nel criterio di costo uniforme hanno un costo pari a n, mentre nel criterio di costo logaritmico hanno un costo pari a $32n$.
In molti casi, infatti, possiamo assumere che gli "elementi" siano rappresentati da un numero costante di bit e che le due misure coincidano a meno di una costante moltiplicativa.
Il criterio che abbiamo utilizzano fin'ora – e che useremo d'ora in poi – è il criterio del costo uniforme, in casi particolari utlizzeremo il criterio di costo logaritmico.

1.3 Misurare il tempo

Consideriamo un'istruzione come elementare se può essere eseguita in tempo "costante" dal processore. Facciamo qualche esempio:

- `a *= 2` effettua un'operazione di shift, è una singola operazione macchina;
- `Math.cos(d)` può essere considerata come un'operazione elementare;
- `min(A, n)` non può essere considerata un'operazione elementare poiché si richiede il minimo di un vettore *arbitrariamente lungo*.

Ma allora cosa come possiamo distinguere in maniera precisa un'operazione elentare da una che non lo è? Abbiamo bisogno di un modello di calcolo, ossia una rappresentazione astratta di un calcolatore. Il quale deve 1. permettere di nascondere i dettagli (tramite astrazione) 2. riflettere la situazione reale (realismo) 3. permettere di trarre conclusioni "formali" sul contesto. La pagina di Wikipedia dei modelli di calcolo può trovare centinaia di modelli di calcolo diversi. La macchina di turing ne è un esempio. É una macchina ideale che manipola – seconodo un insieme prefissato di regole – i dati contenuti su un nastro di lunghezza infinita. Ad ogni passo, la Macchina di Turing:

1. legge il simbolo sotto la testina;
2. modifica il proprio stato interno;
3. scrive il nuovo simbolo nella cella;
4. muove la testina a destra o a sinistra.

Nel corso di laurea magistrale è possibile approfondire questo aspetto, per i nostri scopi questo è un livello di trattazione dell'argomento troppo a basso livello.
Noi utilizzeremo il modello di calcolo RAM, che sta per Random Access Machine. Ossia una macchina che ha una quantità infinita di celle (di dimensione finita) e accesso in tempo costante indipendetemnte dalla posizione (diveramente da ciò che avviene nei nastri); un singolo processore con un set di istruzioni simile a quelli reali i cui costi di esecuzione sono uniformi e ininfluenti ai fini della valutazione (faremo un esempio più avanti).

1.3.1 Calcolo della complessità

Proviamo a calcolare la complessità dell'algoritmo che ricerca il minimo.

```
// calcola il minimo di un vettore arbitrariamente lungo
min(ITEM[] A, int n)
    ITEM min ← A[i]                          Costo: c1     # Volte: 
    da i ← 2 fino a n fai                    Costo: c2     # Volte: n
        se A[i] < min allora                 Costo: c3     # Volte: n − 1
            min ← A[i]                       Costo: c4     # Volte: n − 1
    ritorna min                              Costo: c5     # Volte: 1
```

Ragionamento sul calcolo della complessità L'assegnazione del minimo viene eseguita solo una volta. Il ciclo viene eseguito n volte. l controllo $A[i] < min$ viene eseguito $n-1$ volte in quanto dobbiamo guardare tutto il vettore. Consideriamo *il caso pessimo*, ovvero un vettore ordinato in modo decrescente. L'istruzione di ritorno viene eseguita una volta sola.

Bisogna tenere ben a mente che: ogni istruzione richiede un tempo costante per essere eseguita e viene eseguita un certo no. di volte, dipendente da n, la costante è potenzialmente diversa da istruzione a istruzione.

Sommando tutte le costanti il costo totale risultante è:

$$\begin{aligned} T(n) &= c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 \\ &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) \\ &= an + b \end{aligned}$$

racoogliamo

semplifichiamo

Possiamo quindi notare che le costanti vanno a semplificarsi nei parametri a e b.
Proviamo a calcolare la complessità dell'algoritmo che ricerca un numero intero in un vettore arbitrariamente lungo.

```
// Effettua una ricerca binaria su un vettore di lunghezza arbitraria
binarySearch(ITEM[] A, ITEM v, int i, int j)
```

	Costo	# $i > j$	# $i \leqslant j$
se $i > j$ **allora**	c_1	1	1
	ritorna 0	c_2	1	0
altrimenti			
	int $m \leftarrow \lfloor \frac{(i+j)}{2} \rfloor$	c_3	0	1
	se $A[m] == v$ **allora**	c_4	0	1
		ritorna m	c_5	0	0
	altrimenti se $A[m] < v$ **allora**	c_6	0	1
		ritorna binarySearch$(A, v, m+1, j)$	$c_7 + T\left(\lfloor \frac{n-1}{2} \rfloor\right)$	0	0/1
	altrimenti			
		ritorna binarySearch$(A, v, i, m-1)$	$c_7 + T(\lfloor n/2 \rfloor)$	0	1/0

Ragionamento sul calcolo della complessità Il vettore viene diviso due parti: la parte sinistra di dimensione $\lfloor \frac{n-1}{2} \rfloor$ e la parte destra di dimensione $\lfloor n/2 \rfloor$. Se n è pari allora il vettore viene diviso in due parti uguali, altrimenti il vettore "di destra" avrà un elemento in più. Si andrà cercare sulla metà sinistra o sulla metà destra a seconda che l'elemento cercato sia più grande o più piccolo rispettivamente. Anche in questo caso consideriamo il caso peggiore, ovvero il caso in cui l'elemento non sia presente. Non prendiamo in considerazione il caso fortunato in cui l'elemento che stia cercando sia l'elemento che guardiamo per primo.
Nelle chiamate ricorsive dobbiamo considerare nel costo anche il costo delle sotto chiamate ricorsive con dimensione dell'input pari alla dimesione del vettore passato.

Nota. *Ci è permesso fare questo ragionamento poiché il vettore è ordinato in ordine decrescente.*

Esercizio Cerca nel vettore ordinato l'elemento 0 tramite la procedura binarySearch, calcolando di volta in volta la dimensione del vettore n.

1	2	3	4	5	6	7	8
8	7	6	5	4	3	2	1

Calcolo del caso pessimo Assumiamo per semplicità che

1. n sia una potenza di 2 ($n = 2^k$);
2. l'elemento cercato non sia precisamente e che
3. ad ogni passo scegliamo il vettore di destra (di dimensione $n/2$).

Si scaturiscono due casi:

- il caso base $\boxed{i > j}$, dove $n = 0$ e la relazione di ricorrenza è pari a $T(n) = c_1 + c_2 = c$ dove c è una costante;

- il caso ricorsivo $\boxed{i \leqslant j}$, dove $n > 0$ e dobbiamo tener conto di tutte le costanti moltiplicative $T(n) = T(n/2 + c_1 + c_3 + c_4 + c_6 + c_7)$, raccogliendo le costanti $T(n) = T(n/2) + d$, dove d è la costante che racchiude tutti i costi.

La relazione di ricorrenza che ne segue è la sequente:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n > 0 \end{cases}$$

Nota. *Per calcolare la complessità di una funzione ricorsiva abbiamo bisogno di una funziona di ricorrenza ricorsiva.*

Le equazioni di ricorrenza così fatte $T(n) = e \log(n)$ sono dette a "forma chiusa" e rappresentano la complessità dell'algoritmo.
Risolviamo quindi l'equazione di ricorrenza *tramite espansione*:

$$\begin{aligned} T(n) &= T(n/2) + d && \big(T\left(\tfrac{n}{2} \cdot \tfrac{1}{2}\right) + d\big) + d \\ &= T(n/4) + 2d && \big(T\left(\tfrac{n}{4} \cdot \tfrac{1}{2}\right) + 2d\big) + d \\ &= T(n/8) + 3d \\ &\dots && n = 2^k \implies k = \log n \\ &= T(1) + kd && T(0) = c \\ &= T(0) + (k+1)\,d \\ &= kd + (c+d) && k = \log n \\ &= d \log n + e. \end{aligned}$$

1.4 Ordini di complessità

Per ora, abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due *funzioni di complessità*:

- Ricerca: $T(n) = d \log n + e$, chiamiamo questa funzione **logaritmica** ed utilizzeremo la notazione $\mathcal{O}(\log n)$;
- Minimo: $T(n) = a + b$, chiamiamo questa funzione **lineare** ed utilizzeremo la notazione $\mathcal{O}(n)$.

Abbiamo visto anche una terza funzione che deriva dall'algoritmo banale (*naïf*) per la ricerca del minimo:

- Minimo: $T(n) = fn^2 + gn + h$, chiamiamo questa funzione **quadratica** ed utilizzeremo la notazione $\mathcal{O}(n^2)$.

1.5 Classi di complessità

Tabella 1: Classi di complessità

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	logaritmico
$\sqrt{n}$	3	10	31	100	sublineare
n	10	100	1000	10 000	lineare
$n \log n$	30	664	9965	132 877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^{12}	cubico
2^n	1024	10^{30}	10^{300}	10^{3000}	esponenziale

2 Funzioni di costo, notazione asintotica

Ora andremo a formalizare le nozioni sui limiti superiori ed inferiori che abbiamo accennato in maniera informale nelle lezioni precedenti.

Definizione (Funzione di costo)**.** *Utilizziamo il termine funzione di costo per indicare una funzione* $f: \mathbb{N} \to \mathbb{R}$ *dall'insieme sei numeri naturali ai reali.*

Domanda orale

Definizione (Notazione $\mathcal{O}$)**.** Sia $g(n)$ una funzione di costo; indichiammo con $\mathcal{O}(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists \geqslant 0 : f(n) \leqslant cg(n), \forall n \geqslant m$$

Nota. *Eventuali fattori moltiplicativi non ci interessano.*

La notazione si legge $f(n)$ è "O grande" di $g(n)$, si scrive $f(n) = \mathcal{O}(g(n))$ (questo è un abuso di notazione, dovemmo scrivere $f(n) \in \mathcal{O}(g(n))$, in quanto $\mathcal{O}$ è un insieme (una famiglia di funzioni), ma è diventato uso comune questa notazione poiché si può fare una specie di aritmetica sopra) e sta a significare che $g(n)$ è un limite asintotico superiore per $f(n)$, ossia che $f(n)$ cresce al più (al massimo) come $g(n)$.

Definizione (Notazione Ω)**.** *Sia* $g(n)$ *una funzione di costo; indichiammo con* $\Omega(g(n))$ *l'insieme delle funzioni* $f(n)$ *tali per cui:*

$$\exists c > 0, \exists \geqslant 0 : f(n) \geqslant cg(n), \forall n \geqslant m$$

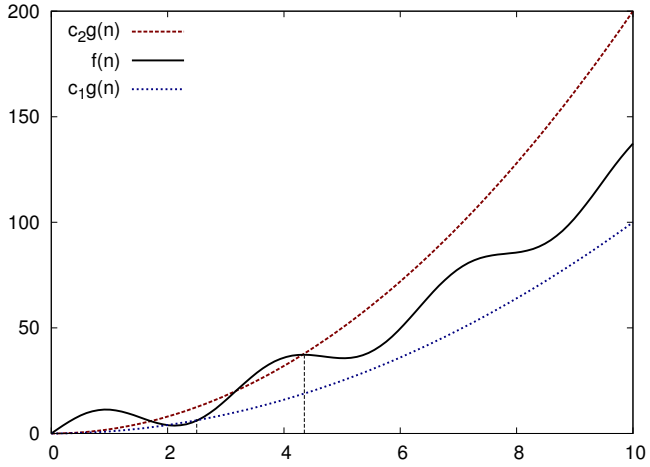
La notazione si legge $f(n)$ è "Omega grande" (nella letteratura big-O) di $g(n)$, si scrive $f(n) = \Omega(g(n))$ e sta a significare che $g(n)$ è un limite asintotico inferiore per $f(n)$, ossia che $f(n)$ cresce almeno quanto (non di meno) come $g(n)$.

Definizione (Notazione Θ)**.** *Sia* $g(n)$ *una funzione di costo; indichiammo con* $\Theta g(n)$ *l'insieme delle funzioni* $f(n)$ *tali per cui:*

$$\exists c > 0, \exists \geqslant 0 : c_1 g(n) \leqslant f(n) \leqslant c_2 g(n), \forall n \geqslant m$$

La notazione si legge $f(n)$ è "Theta" di $g(n)$, si scrive $f(n) = \Theta g(n)$ e sta a significare che $f(n)$ cresce *esattamente* come $g(n)$ aldilà di fattori moltiplicativi, $f(n) = \Theta g(n)$ avviene se e solo se $f(n) = \mathcal{O}g(n)$ e $f(n) = \Theta g(n)$.

Figura 1: Notazione asintotica



Definizione (Complessità in tempo di un algoritmo)**.** *La più grande quantità di tempo richiesta per un input di dimensione* n*.*

- $\mathcal{O}(f(n))$: per tutti gli input, l'algoritmo costa al più $f(n)$;
- $\Omega(f(n))$: per tutti gli input, l'algoritmo costa almeno $f(n)$;
- $\Theta(f(n))$: l'algoritmo richiede $\Theta(f(n))$ per tutti gli input.

Definizione (Complessità in tempo di un problema computazionale)**.** *La complessità in tempo relative a tutte le possibili soluzioni.*

- $\mathcal{O}(f(n))$: complessità del miglior algoritmo che risolve il problema;
- $\Omega(f(n))$: dimostrare che nessun algoritmo può risolvere il problema in tempo inferiore a $\Omega(f(n))$;
- $\Theta(f(n))$: abbiamo trovato l'algoritmo ottimo.

2.1 Esercizi

Iniziamo con gli esercizi banali che ci permettono di introdurre delle tecniche che utilizzeremo con le ricorrenze. In particolare ci servono a renderci conto che non stiamo dimostrando equazioni, ma bensì disequazioni.

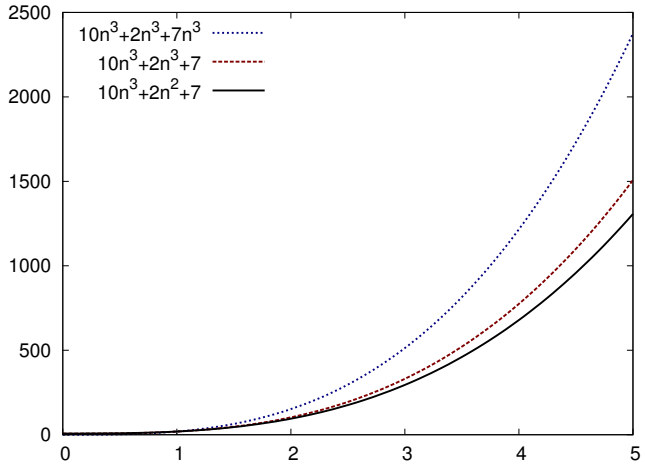
$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 \\
&\leqslant 10n^3 + 2n^3 + 7 && \forall n \geqslant 1 \\
&\leqslant 10n^3 + 2n^3 + 7n^3 && \textit{sommiamo i termini} \\
&= 19n^3 && \textit{esiste una certa costante } c \\
&\stackrel{?}{\leqslant} cn^3 && \textit{per la quale } f(n) \leqslant cn^3 \ ? \\
19n^3 &\leqslant cn^3 && \textit{metto a confronto} \\
19\cancel{n^3} &\leqslant c\cancel{n^3} && \textit{semplifico}
\end{aligned}$$

che è vera per ogni $c \geqslant 19$ (abbiamo così trovato la costante moltiplicativa) e per ogni $n \geqslant 1$ (introdotta nei calcoli), quindi $m = 1$.

Figura 2: Risluzione grafica dell'esercizio



Nota. *In generale noi considereremo solo valori di n positivi, in quanto le funzione di costo sono definite sull'insieme dei numeri naturali, non ha alcun senso definire una funzione di costo su una dimensione dell'input negativa.*

Nota. *Dato lo stesso esercizio posso esserci passaggi risolutivi diversi.*

Risolviamo l'esercizio precedente diversamente.

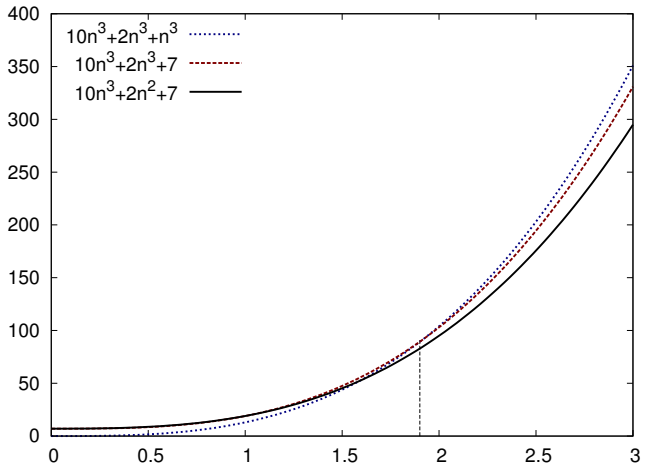
$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 && \text{(} \forall n \geqslant 1 \text{)} \\
&\leqslant 10n^3 + 2n^3 + 7 && \text{(} \forall n \geqslant \sqrt[3]{7} \text{)} \\
&\leqslant 10n^3 + 2n^3 + n^3 && \text{(sommiamo i termini)} \\
&= 13n^3 && \text{(esiste una certa costante } c \\
&\stackrel{?}{\leqslant} cn^3 && \text{per la quale } f(n) \leqslant cn^3 \text{ ?)} \\
13n^3 &\leqslant cn^3 && \text{(metto a confronto)} \\
13\cancel{n^3} &\leqslant c\cancel{n^3} && \text{(semplifico)}
\end{aligned}$$

che è vera per ogni $c \geqslant 13$ e per ogni $n \geqslant \sqrt[3]{7}$ (ad esempio con $n = 2$ abbiamo $n^3 = 2^3 = 8$ che soddisfa la nostra condizione), quindi usiamo $m = 2$ (abbiamo semplificato, sarebbe $m = \sqrt[3]{7}$, ma possiamo prendere un qualunque valore che si trova dopo in modo totalmente arbitrario).

Figura 3: Risluzione grafica dell'esercizio



$$\boxed{f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)}$$

Limite inferiore: Dobbiamo dimostrare che $\exists c_1 > 0, \exists m_1 \geqslant 0 : f(n) \geqslant c_1 n^2, \forall n \geqslant m_1$

$$\begin{aligned}
f(n) &= 3n^2 + 7n && \text{(} \forall n \geqslant 0 \text{)} \\
&\geqslant 3n^2 && \text{(esiste una certa costante } c \\
&\stackrel{?}{\geqslant} c_1 n^2 && \text{per la quale } f(n) \leqslant c_1 n^2 \text{ ?)} \\
3n^2 &\leqslant c_1 n^2 && \text{(metto a confronto)} \\
3\cancel{n^2} &\leqslant c_1\cancel{n^2} && \text{(semplifico)}
\end{aligned}$$

che è vera per ogni $c_1 \leqslant 3$ e per ogni $n \geqslant 0$ (introdotta nei calcoli), quindi $m_1 = 0$.

Nota. *Abbiamo dimostrato quindi che* $f(n) = \Omega(n^2)$

Limite superiore: Dobbiamo dimostrare che $\exists c_2 \geqslant 0, \exists m_2 \geqslant 0 : f(n) \leqslant c_2 n^2, \forall n \geqslant m_2$

$$
\begin{aligned}
f(n) &= 3n^2 + 2n^2 + 7n && \text{per } \forall n \geqslant 1 \\
&\leqslant 3n^2 + 7n^2 && \text{raccogliamo} \\
&\leqslant 10n^2 && \text{esiste una certa costante } c \\
&\overset{?}{\leqslant} c_2 n^2 && \text{per la quale } f(n) \leqslant c_2 n^2 \ ? \\
10n^2 &\leqslant c_2 n^2 && \text{metto a confronto} \\
10\cancel{n^2} &\leqslant c_2 \cancel{n^2} && \text{semplifico}
\end{aligned}
$$

che è vera per ogni $c_2 \geqslant 10$ e per ogni $n \geqslant 1$, quindi $m_2 = 1$.

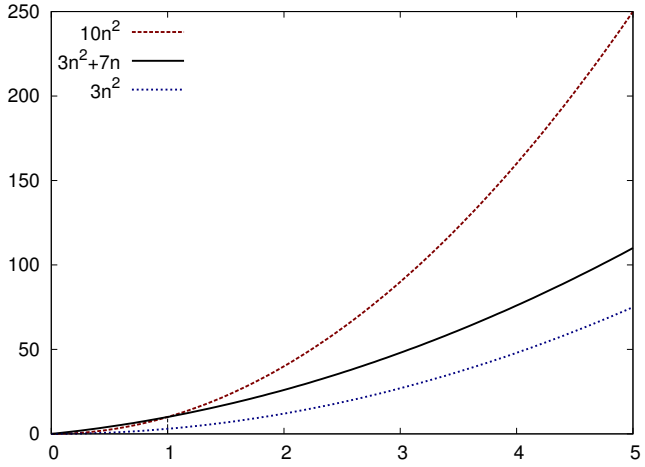
Nota. *Abbiamo dimostrato quindi che* $f(n) = \mathcal{O}(n^2)$

Notazione Θ: $\exists c_1 > 0, \exists c_2 > 0, \exists m \geqslant 0 : c_1 n^2 \leqslant f(n) \leqslant c_2 n^2, \forall n \geqslant m$.
Con questi paramentri:

- $c_1 = 3$
- $c_1 = 10$
- $m = max\{m_1, m_2\} = max\{0, 1\} = 1$, ossia un valore dopo il quale la nostra proprietà è provata

Nota. *Abbiamo dimostrato quindi che* $f(n) = \Theta(n^2)$

Figura 4: Risluzione grafica dell'esercizio

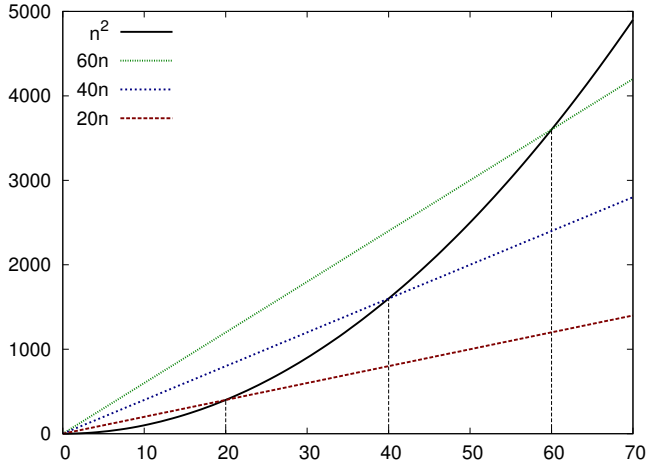


Errori comuni durante la risoluzione gli esercizi

$$\boxed{f(n) = n^2 \overset{?}{=} \mathcal{O}(n)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m > 0 : n^2 \geqslant cn, \forall n \geqslant m$.
Otteniamo che $n^2 \leqslant cn \Leftrightarrow c \geqslant n$, questo significa che c cresce con il crescere di n, ovvero che non possiamo scegliere una costante c.

Figura 5: Per qualunque fattore c (la pendenza) scegliamo la curva quadratica crescerà sempre più velocemente da un punto in poi



$$f(n) = n^2 \stackrel{?}{=} \Omega(n^3)$$

Limite inferiore: Dobbiamo dimostrare che $\exists c > 0, \exists m > 0, n^2 \geqslant cn^3, \forall n \geqslant m$.
Otteniamo che $n^2 \geqslant cn^3 \Leftrightarrow c \leqslant \frac{1}{n}$, questo significa che c diminuisce al crescere di n, ovvero che non possiamo scegliere una costante c.

2.2 Complessità degli algoritmi e dei problemi a confronto

In questa sezione ragioneremo su alcune soluzioni che ci sono state insegnate, in alcuni casi si può migliorare la complessità, in altri è impossibile fare di meglio.

Qual è il rapporto fra un problema computazionale e l'algoritmo?

2.2.1 Moltiplicare numeri complessi

Domanda orale

La moltiplicazione fra numeri complessi avviene nel seguente modo: $(a+bi)(c+di) = [ac-db] + [ad+bc]i$.
Abbiamo in input a, b, c, d e dobbiamo restituire in output $ac-bd$ e $ad+bc$.

Consideriamo un modello di calcolo dove la moltiplicazione costa 1 e le addizioni e sottrazioni costano 0.01.

1. Quanto costa l'algortmo dettato dalla definizione?
2. Riesci a fare meglio di così?
3. Qual è il ruolo del modello di calcolo?

L'algoritmo banale dettato dalla definizione costa 4.02, in quanto bisogna fare 4 moltiplicazioni, 1 somma ed una sottrazione.

La seguente è la soluzione di Gauss al problema, datata 1805.
Input: a, b, c, d, Output: $A1 = ac - bd$, $A2 = ad + bc$

$$\begin{aligned} m_1 &= a \times c \\ m_2 &= b \times d \\ A_1 &= m_1 - m_2 \\ m_3 &= (a+b)\cdot(c+d) = ac + ad + bc + bd \\ A_2 &= m_3 - m_1 - m_2 = ad + bc \end{aligned}$$

calcolo i valori intermedi

evito una moltiplicazione

Il costo totale è 3.05.

Si può fare ancora meglio di così? Oppure, è possibile dimostrare che non si può fare di meglio?

2.2.2 Sommare numeri binari

L'algoritmo elementare della somma richiedere di esaminare tutti gli n bit, il costo totale risulta cn, dove c è il costo per sommare tre bit e generare il riporto.

Esiste un metodo più efficiente?

É dimostrabile per assurdo che *non è possibile fare di meglio* di una soluzione lineare, poiché non è possibile sommare due numeri binari senza guardare tutti gli n bit.

Limite superiore alla complessità di un problema

Notazione ($\mathcal{O}(f(n))$ — Limite superiore)**.** *Un problema ha complessità* $\mathcal{O}(f(n))$ se esiste almeno un algoritmo *che ha complessità* $\mathcal{O}(f(n))$.

Nota. *Il problema della somma dei numeri binari ha complessità* $\mathcal{O}(n)$.

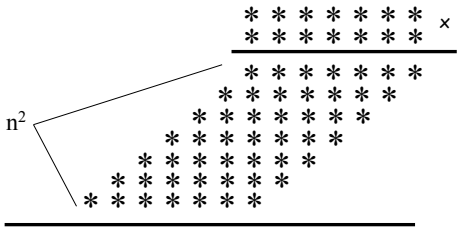
Definizione ($\Omega(f(n))$ — Limite inferiore)**.** *Un problema ha complessità* $\Omega(f(n))$ se tutti i possibili algoritmi *che lo risolvono hanno complessità* $\Omega(f(n))$.

Nota. *Il problema della somma dei numeri binari ha complessità* $\Omega(n)$.

2.2.3 Moltiplicare numeri binari

L'algoritmo elementare del prodotto richiede di moltiplicare ogni bit con ogni altro bit, per un costo totale di cn^2

Figura 6: Moltiplicazione di due numeri binari



Si potrebbe concludere che il problema della moltiplicazione è inerentemente più costoso del problema dell'addizione, ne è la conferma la nostra esperienza.

Nota. *Per provare che il problema del prodotto è più costoso del problema della somma, dobbiamo provare che non esiste una soluzione in tempo lineare del prodotto.*

Abbiamo infatti erroneamente confrontato gli algoritmi, non i problemi! Sappiamo solo che l'algoritmo che ci hanno insegnato della somma è più efficiente di quello della moltiplicazione.

Nel 1960, Kolmogorov enunciò che la moltiplicazione avesse limite inferiore pari a $\Omega(n^2)$, una settimana dopo un suo studente Karatsuba riuscì a provare il contrario. Andiamo a vedere la sua soluzione.

Moltiplicazione di Karatsuba

Domanda orale

Karatsuba ebbe un approccio dividi-et-impera.

Definizione (Approccio dividi-et-impera)**.** *Si svolge in tre parti:*

- *Dividi: dividi il problema in sottoproblemi di dimensioni inferiori;*
- *Impera: risolvi i sottoproblemi in maniera ricorsiva;*
- *Combina: unisci le soluzioni dei sottoproblemi in modo da ottenere la risposta del problema principale.*

$$X = a \cdot 2^{n/2} + b$$
$$Y = c \cdot 2^{n/2} + d$$
$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + db$$

```
// moltiplica due numeri binari
bool[ ] pdi(bool[ ] X, bool[ ] Y, int n)
    // X:  numero binario
    // Y:  numero binario
    // n:  numero di bit contenuti
    se n == 1 allora
        ritorna X[1] · Y[1] // eseguo la moltiplicazione di due bit
    allora
        spezza X in a;b e Y in c;d
        ritorna pdi(a, c, n/2) · 2^n + (pdi(a, d, n/2) + pdi(b, c, n/2)) · 2^(n/2) + pdi(b, d, n/2)
```

Complessità Moltiplicare per 2^t è pari ad eseguire uno shift di t posizioni, in tempo lineare, quindi l'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Non sappiamo ancora trattare questo genere di problemi, facciamo solo degli accenni.

Analisi della ricorsione (vedi figura) Al primo passo la chiamata ricorsiva viene viene su una dimesione n, al secondo passo vengono effettuate 4 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $4^2 = 16$ chiamate ricorsive su una dimesione $n/2^2 \ldots$ al livello i-esimo vengono effettuate 4^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $4^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $4^{\log n} = n^{\log 4} = n^2$, le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^2)$. É possibile ridurre la complessità.

$$A_1 = a \times c$$
$$A_2 = b \times d$$
$$m = (a + b) \times (c + d) = ac + ad + bc + bd$$
$$A_3 = m - A_1 - A_2 = ad + bc$$

calcolo un valore intermedio

evito una moltiplicazione

Principio Effettuo un unica moltiplicazione che mi permette di calcolare un valore intemedio che contiene la somma di tutte le combinazioni, e ricavo $ad + bc$ tramite due sottrazioni (nello stesso modo in cui lavorava Gauss), evitando così una moltiplicazione.

```
bool[ ] karatsuba(bool[ ] X, bool[ ] Y, int n)
    se n == 1 allora
        ritorna X[1] · Y[1] // rimane invariato
    altrimenti
        spezza X in a;b e Y in c;d
        bool[ ] A1 = karatsuba(a, c, n/2)
        bool[ ] A3 = karatsuba(b, d, n/2)
        bool[ ] m = karatsuba(a+b, c+d, n/2) // potrebbe essere n/2 + 1
        bool[ ] A2 = m − A1 − A3 // ottengo A2 tramite sottrazione
        ritorna A1 · 2^n + A2 · 2^(n/2) + A3 // effettuo degli shift
```

Complessità L'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} c_1 & n = 1 \\ 3T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Analisi della ricorsione (vedi figura) Al primo passo la chiamata ricorsiva viene viene su una dimesione n, al secondo passo vengono effettuate 3 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $4^2 = 16$ chiamate ricorsive su una dimesione $n/3^2 \ldots$ al livello i-esimo vengono effettuate 3^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $3^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $3^{\log n} = n^{\log 3} = n^{1.58\ldots}$, le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^{1.58\ldots})$.

Nota. *L'algoritmo ingenuo (naïf) non è sempre il migliore a meno che non sia possibile dimostrare il contrario.*

Negli anni sono stati proposti diversi algoritmi, che il limite inferiore al problema della moltiplicazione sia $\Omega(n \log n)$ è una congettura. Una congettura è un'affermazione o un giudizio fondato sull'intuito, ritenuto probabilmente vero, ma non ancora rigorosamente dimostrato, cioè dunque relegato solamente a rango di ipotesi.
Nella GNU Multiple Precision Arithmetic Library vengono utilizzati diversi algoritmi al cresce di n, il valore soglia per cui si predilige un algoritmo rispetto ad un altro dipende dal tipo di architettura.

2.3 Algoritmi di ordinamento

In questa lezione impareremo a capire quando è meglio utilizzare un algoritmo di ordinamento rispetto ad un altro.
In alcuni casi, gli algoritmi si comportano diversamente a seconda delle caratteristiche dell'input. Conoscere in anticipo tali caratteristiche permette di scegliere il miglio algoritmo in quella situazione

Tipologia di analisi

Esistono tre tipi di analisi:

1. Analisi del caso pessimo: è la tipologia più importante, il tempo di esecuzione nel caso peggiore è il limite superiore al tempo di esecuzione per qualsiasi input. Per alcuni algoritmi il caso peggiore si verifica molto spesso (ad esempio nella ricerca di dati non presenti nel database);
2. Analisi del caso medio: è difficile da definire (cosa si intende per "medio"?), dobbiamo avere una conoscenza pregressa sulle distribuzioni
3. Analisi del caso ottimo: può avere senso se si conoscono informazioni particolari sull'input.

Problema dell'ordinamento

Il problema dell'ordinamento consiste nell'avere una sequenza $A = a_1, a_2, \ldots, a_n$ di n valori in input e di restituire in output una sequenza $B = b_1, b_2, \ldots, b_n$ che sia una permutazione di A e tale per cui $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$, ovvero che ci sia un ordinamento totale.
Un approccio "demente" è quello di generare tutte le possibili permutazioni (complessità $n!$) fino a quando non ne trovo una ordinata.

2.3.1 Selection sort

Un approccio banale (naïf) è quello di cercare il minimo e metterlo nella posizione corretta, riducendo il problema agli $n-1$ valori restanti.

```
// effettua l'ordinamento di un vettore
selectionSort(ITEM[] A, int n)
    da int i ← 1 fino a n fai
        int j ← min(A, i, n) // ricerca il nuovo minimo
        swap(A[i], A[j]) // lo metto nella posizione corretta

// cerca l'indice dell'elemento più piccolo
int min(ITEM[] A, int i, int j)
    int min ← k // posizione del minimo parziale
    da int h ← k + 1 fino a n fai
        se A[h] < A[min] allora // ho trovato un nuovo minimo
            min ← h // nuovo minimo parziale

    ritorna min // restituisco l'indice dell'elemento più piccolo
```

Suggerimento. *Provalo su carta! Un algoritmo per essere capito dev'essere provato!*

Analisi della complessità Il ciclo effettua n, $n-1$, $\ldots$, 2 chiamate della funzione min che viene effettuata su una dimensione di $n-1$ che via via con l'esecuzione dell'algoritmo diminuisce.

$$
\begin{aligned}
&\textstyle\sum_{i=1}^{n-1}(n-1) && \text{)} \; 10+9+\cdots+1 \Leftrightarrow 1+2+\cdots+10\\
&= \textstyle\sum_{i=1}^{n-1} i \\
&= \tfrac{n(n-1)}{2} && \text{)} \; \textit{svolgo i calcoli}\\
&= n^2 - \tfrac{n}{2} \\
&= \Theta(n^2)
\end{aligned}
$$

Posso dire che è $\Theta(n^2)$, e non solo che $\Omega(n^2)$, perché indipendetemnte dall'ordine dei numeri ci metterà sempre lo stesso tempo.

2.3.2 Insertion sort

Un algoritmo che si basa sul principio di ordinamento di una "mano" di carte da gioco è il seguente:

```
// effettua l'ordinamento di un vettore
insertionSort(ITEM[] A, int n)
    da int i = 2 fino a n fai // il 1° elemento è ordinato
        ITEM temp ← A[i] // elemento da ordinare
        int j ← i

        finché j > i and A[j − 1] fai
            A[j] ← A[j − 1] // copio l'elemento
            j −− // mi sposto

        A[j] ← temp
```

Questo è un algoritmo molto efficiente per ordinare piccoli insiemi di elementi.

Analisi della complessità Il costo di esecuzione di questo algoritmo non dipende solo dalla sua dimensione, ma anche dalla distribuzione dei dati in ingresso. Nel caso in cui il vettore sia *già ordinato* il costo è $\mathcal{O}(n)$, in quanto non entro mai nel secondo ciclo in quando la condizione risulta falsa. Nel caso in cui il vettore sia *ordinato in ordine inverso* è $\Omega(n^2)$. In media (informalmente) possiamo assumere che metà dei

valori sia molto di molto rispetto la loro disposizione finale e quindi metà di loro dovrannno fare n passi per arrivare alla destinazione per una complessità di $n \cdot n/2 = \mathcal{O}(n^2)$.
Quindi quando sappiamo che i valori sono quasi ordinati o che n è molto piccolo — nell'ordine di 16 o 32 — allora questo algoritmo risulta efficiente.

2.3.3 Merge sort

MergeSort è basato sulla tecnica dividi-et-impera vista in precendenza. Ma come la utilizza?

Definizione 2.1 (Approccio dividi-et-impera di MergeSort)**.** *Si svolge in tre parti:*

- *Dividi: Spezza il vettore di n elementi in 2 sottovettori di $\frac{n}{2}$ elementi;*
- *Impera: Chiama* mergeSort *ricorsivamente sui due sottovettori (ottenendo due metà ordinate);*
- *Combina: Unisci le due sequenze ordinate (`merge`).*

Si sfrutta il fatto che due sottovettori sono già ordinati per ordinare più velocemente.

```
mergeSort(ITEM[] A, int primo, int ultimo)
    se primo < ultimo allora // devono esistere almeno due elementi
        int mezzo ← ⌊(primo+ultimo)/2⌋
        mergeSort(A, primo, mezzo)
        mergeSort(A, mezzo+1, ultimo)
        merge(A, primo, ultimo, mezzo)
merge(ITEM A, int primo, int ultimo, int mezzo)
    int i, j, k, h
    // inizializzo i puntatori
    i ← primo j ← mezzo k ← primo
    // k:  indica la prossima posizione di scrittura
    // fintanto che entravi
    finché i ⩽ mezzo and j ⩽ ultimo fai
        se A[i] ⩽ A[j] allora
            // l'elemento è gia ordinato
            B[k] ← A[i]
            i++
        altrimenti
            B[k] ← A[j]
            j++
        // in entrambi i casi ho inserito un valore
        k++
    // se uno dei due vettori finisce ricopio la parte ordinata alla fine del vettore
       d'appoggio
    j ← ultimo
    da h ← mezzo fino a i fai
        A[j] ← A[h]
        j −−
    // ricopio il vettore d'appoggio del vettore originale
    da j ← primo fino a k − 1 fai
        A[j] ← B[j]
```

Analisi della complessità Assumiamo (per semplicità) che $n = 2^k$, ovvero che l'altezza dell'albero di suddivisioni sia esattamente $k = \log_2 n$ e che tutti i sottovettori abbiano dimensioni che sono potenze esatte di 2. L'equazione di ricorrenza risultante è la seguente:

$$T = \begin{cases} \Theta(1) & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + \Theta(n) & n > 1 \end{cases}$$
$$= \begin{cases} c & n = 1 \\ 2T(n/2) + dn & n > 1 \end{cases}$$

Qual è il costo computazionale di mergeSort?

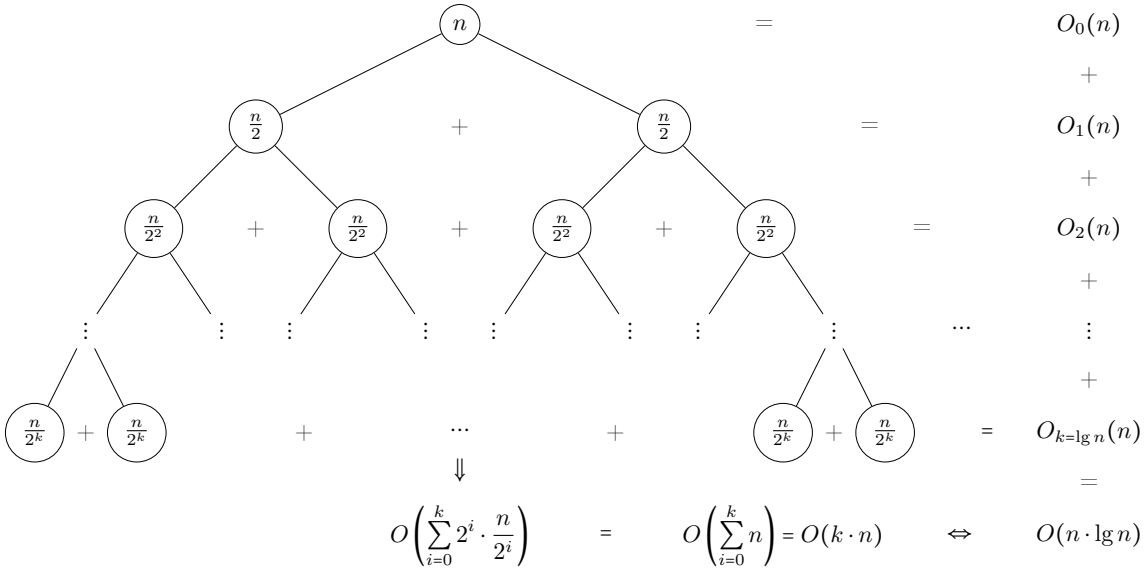


Figura 7: Analisi per livelli

L'analisi per livelli è la seguente:

$$\begin{aligned} & \mathcal{O}\left(\textstyle\sum_{i=0}^{k} 2^i \frac{n}{2^i}\right) \\ &= \mathcal{O}\left(\textstyle\sum_{i=0}^{k} n\right) && \text{semplifico} \\ &= \mathcal{O}(k \cdot n) && \text{equivalente} \\ &= \mathcal{O}(n \log n) && k = \log n \end{aligned}$$

$\mathcal{O}(n \log n)$ è asintoticamente migliore di $\mathcal{O}(n^2)$. Questo algoritmo è preferibile — per grandi dimensioni di n — al selectionSort e all'insertionSort.