FONDAMENTI DI INFORMATICA

DECIMA LEZIONE

Prof. Alfredo Accattatis

(accattatis@ing.uniroma2.it)

Tutor: Prof. Vittorio Calafiore

(vcalafio@gmail.com)

Complessità computazionale

- Un problema può essere risolto utilizzando diversi algoritmi
- Qual è il migliore?

- => la bontà o efficienza di un algoritmo si misura in base alla quantità minima di risorse sufficienti per il calcolo; queste sono:
  - Il tempo necessario per eseguire le azioni elementari (complessità temporale)
  - Lo spazio necessario per memorizzare e manipolare i dati (complessità spaziale)

Complessità computazionale

- Faremo riferimento, per motivi di semplicità, unicamente al concetto di complessità temporale.
- In realtà la complessità e un fenomeno….complesso, che dovrebbe tenere conto di moltissimi fattori
- Per esempio lo stesso concetto di ‘spazio’ dovrebbe tenere conto delle differenti tipologie di spazio (memoria) esistenti e di come le diverse architetture la impiegano, compreso il relativo grado di parallelismo
- In questo corso ci interessa chiarire solo alcune linee essenziali

Il tempo… risorsa di maggior interesse

- Non è significativo misurare il tempo di calcolo di un algoritmo in base al numero di secondi richiesti per l'esecuzione su un elaboratore di un programma che lo descriva. Infatti, tale tempo dipende da:
  - i dati di ingresso (non solo dimensioni…)
  - il linguaggio in cui è scritto il programma
  - la qualità della traduzione in sequenze di bit
  - la velocità dell'elaboratore
  - la tipologia del microprocessore utilizzato (RISC, CISC, DSP etc.) ossia la sua architettura (Neumann, Harvard etc.)

Tempo di esecuzione

- Una buona misura dell'efficienza di un algoritmo deve prescindere dal calcolatore utilizzato (quindi dalla tipologia del microprocessore, frequenza di clock ma anche periferiche aggiuntive etc.)

- Occorre una misura "astratta" che tenga conto (solo, o principalmente) del "metodo di risoluzione" con cui l'algoritmo effettua la computazione

Tempo di esecuzione: PASSI BASE

- Per calcolare il tempo di esecuzione di un programma abbiamo bisogno di:
  - *Valutare tutte le operazioni «atomiche» che verranno eseguite dal programma data un'istanza del programma e valutare, in una fase successiva, quanto ogni operazione impiega*
- In pratica, quando effettuiamo la stima:
  - Selezioniamo e contiamo solo un sottoinsieme di operazioni (che consideriamo significative)
  - Assumiamo che ognuna impieghi lo stesso tempo
  - Sono i PASSI BASE

Passi base

Sono il numero complessivo di operazioni elementari in funzione della dimensione «n» dei dati in ingresso

- Le operazioni elementari sono quelle:
  - aritmetiche
  - logiche
  - di confronto
  - di assegnazione

- Solitamente, il numero di operazioni considerate è valutato nel **caso peggiore**, ossia nel caso di dati in ingresso più *sfavorevoli* tra tutti quelli di dimensione n. Inoltre come detto le consideriamo «equitemporali»

Come calcolare i passi base:

- Operazioni atomiche hanno costo 1;
- Il costo di una sequenza è la somma dei costi individuali;
- Il costo di un *loop* è la somma dei singoli passi, incluso il costo del "test" della condizione di fine;
- Istruzioni che usano array hanno un costo che si calcola espandendoli nel loro "loop equivalente";
- Una istruzione condizionale ha un costo "worst case" che è il più grande tra la parte "if" e la parte "else"; per ottenere il "costo medio" moltiplicheremo i due rami per la probabilità che hanno di essere percorsi; a cui aggiungeremo il costo dell'operazione di selezione.

Calcolare il costo: esempi

Passi base

Quantità scalari.

```
a = 2.5;                    % 0 o 1: in genere ignorato tranne quantità;
b = a * a + 1;              % 2 operazioni atomiche ;
c = b^3;                    % b^3 è b*b*b , ancora 2 operazioni atomiche ;
for k = n1 : n2             % Eseguito in ( n2 – n1 + 1 ) passi
   c = a + b
end                           % costo totale : 1*( n2 – n1 + 1 )
if ( mod( k , 2 ) == 0 )    % se k è un intero casuale 50% di prob.
      c = a * b + c ;       % “worst case” se ramo IF di costo 2
 else                       % costo medio 1.5
       b = b + 1;           % più 2 per valutare ( MOD() == 0 )
end
```

Esempio 1

Esercizio 4

È data la seguente funzione Matlab

```
function [x n]=simple(a,b)
  n=length(b);
  x = b;
  for i=1:n
    d = diag(a);
    d = d./2;
    a = a + sum(d);
    x = x + d;
 end
```

Si stimi il numero di operazioni aritmetiche eseguite, assumendo che a sia una matrice $n \times n$ e che b sia un array $n \times 1$.

Soluzione 1

```
function [x n]=simple(a,b)
   n=length(b);
   x = b;
   for i=1:n    % ciclo eseguito n volte
      d = diag(a);
      d = d./2;          % n operazioni aritmetiche
      a = a + sum(d); % n^2 +n operazioni aritmetiche
      x = x + d;         % n operazioni aritmetiche
   end
```

in totale vengono eseguite $n(n^2 + 3n)$ operazioni aritmetiche

Esempio 2

Esercizio 2

È data la seguente funzione Matlab

```
function [x]=simple(a)
        for i=1:size(a,1)-1
                d = mean(a);
                d = d.*2;
                a = a + sum(a*d');
        end
        x=a;
```

Si stimi il numero di operazioni aritmetiche eseguite, assumendo che a sia una matrice $n \times n$ e che n sia un intero positivo.

Soluzione 2

Soluzione Esercizio 2

Il numero di operazioni aritmetiche è così calcolato:

- `d = mean(a)` richiede $n * n$ somme e n divisioni
- `d = d.*2` richiede n moltiplicazioni
- `a = a + sum(a*d')` richiede $n * n$ moltiplicazioni e $n * n$ addizioni per calcolare `sum(a*d')` e $n * n$ somme per calcolare il valore finale di `a`

Il numero di operazioni su descritte è eseguito $n - 1$ volte. Quindi abbiamo in totale $(n-1)(4n^2 + 2n) = 4n^3 - 2n^2 - 2n$ operazioni aritmetiche.

Calcolare il costo

Gli «O» grandi

«O» grandi

- Una funzione f(n) è di ordine g(n) e si scrive

$$f(n) = O(g(n))$$

Se esiste una costante numerica **C** maggiore di zero tale che valga (eccetto per un numero finito di valori di n):

$f(n) <= C*g(n)$ ossia $f(n)/g(n) <= C$

La funzione è LIMITATA ossia al tendere di n all'infinito

Il rapporto AL PIU' assumerà il valore **C**

«O» grandi

Usando la definizione di ordine “O”, potremo in effetti dire che la funzione 3*n + 7 è sicuramente di ordine “n”, infatti:

3*n + 7 = O(n) giacché esiste una costante C tale che:

3*n + 7 <= C*n per ogni “n”.

Infatti se scegliamo C = 11 avremo sicuramente (verificate):

3*n + 7 <= 11*n per ogni “n” (tranne 0)

Ma vale anche che:

3*n + 7 <= n^2 per n>=6

«O» grandi

Quindi secondo la definizione appena data la nostra 3*n+7 è anche di ordine n^2 e si può vedere che è anche di ordine 2^n e così via. Qualcosa non torna.

Notiamo però che il viceversa vale SOLO per un caso, ossia:

$n = O(3*n+7)$

ma non vale:

$n^2 = O(3*n + 7)$

Infatti $n^2/(3*n + 7)$ NON è limitata (se "n" tende ad infinito il rapporto tende a infinito).

«O» grandi

Adesso possiamo finalmente definire la *Complessità Asintotica*

Si dice che f(n) ha complessità asintotica g(n) se vale:

$f(n) = O(g(n))$

E g(n) è la *più piccola* delle funzioni che rispettano la definizione sopra

In questo modo definiamo implicitamente una relazione di ORDINE tra le funzioni

«O» grandi

$4n + 5$ è di complessità “ n ”

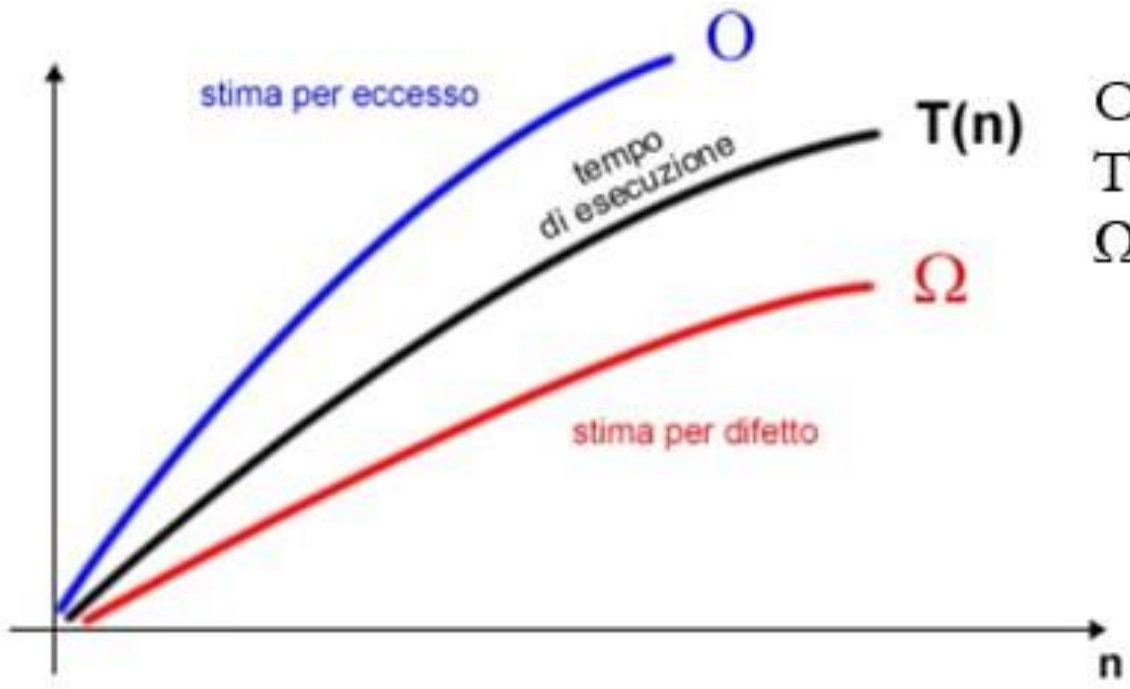
$3n^3 + n - 2$ è di complessità “ n^3 ”

$\log(n) + n$ è di complessità “ n”

Come risulta evidente, la complessità è definita dal termine di complessità maggiore.

Attenzione! Trascurare i termini di ordine inferiore significa dire che un algoritmo la cui complessità in passi base è $10n^2 + n + 12$ è di ordine n^2 al pari di uno eseguito con passi totali $n^2 + 3$. Ma il secondo, quando implementato su una macchina reale può impiegare ore mentre il primo qualche giorno.

«O» grandi



O: stima per eccesso
T(n) : passi base
Ω : stima per difetto

«O» grandi

- $2n + 5$ — complessità asintotica n (lineare)
- $3n^2 + 5n$ — complessità asintotica n^2 (quadratica)
- $4 * 2^n$ — complessità asintotica 2^n (esponenziale di base 2)
- $3n + 2^n + 5^n$ — complessità asintotica 5^n (esponenziale di base 5)
- $2^n + 5n + n! + 5^n$ — complessità asintotica $n^n \sqrt{n}$ (fattoriale)

Formalismo degli «O» grandi

- Perché dunque usare il concetto di “O grande” che potrebbe portarci ad errori di valutazione?

- Perché grazie a questo concetto ed alle operazioni che presto vedremo definite sugli “O grandi” (algebra degli “O grandi”) potremo calcolare la complessità di qualsiasi algoritmo anche molto sofisticato senza conoscerne la complessità in passi base (questi ultimi possono essere a volte difficilissimi da valutare).

Algebra degli «O» grandi

(1)Blocchi di codice in sequenza

(2)Blocchi di codice annidati

(1) Nel primo caso detta $g_1(n)$ la complessità del primo blocco e $g_2(n)$ la complessità del secondo si ha:

$O(g1(n) + g2(n)) = O(max\{ g1(n), g2(n) \})$

Quindi “vince” il blocco a complessità maggiore.

Algebra degli «O» grandi

(2) Nel secondo caso detta $g_1(n)$ la complessità del blocco esterno e $g_2(n)$ la complessità del blocco interno si ha:

$$O(g1(n) * g2(n)) = O(g1(n)) * O(g2(n))$$

Ossia la complessità è data dal prodotto della complessità dei blocchi annidati.

Tempo di esecuzione

- NOTATE che il tempo effettivo di esecuzione non è necessariamente lo stesso per tutte le istanze di taglia ‘n’: abbiamo definito implicitamente il caso peggiore di complessità, richiedendo l’esistenza di un limite superiore valido per tutte le istanze. Si potrebbe anche definire una complessità media su tutte le istanze con la stessa taglia ‘n’ ed il caso migliore.
- ESEMPIO: sommare due vettori di taglia ‘n’ richiede sempre n operazioni aritmetiche, ma ordinare un vettore di n componenti dipende dal contenuto del vettore, ossia dai valori delle ‘n’ componenti. Pensate ad un vettore già ***ordinato*** oppure ***contro-ordinato***.

```
SELECTIONSORT(A, n)
1: per i = 1,2,...,n ripeti
2:     per j = i + 1,...,n ripeti
3:         se a_j < a_i allora
4:             scambia a_i e a_j
5:         fine-condizione
6:     fine-ciclo
7: fine-ciclo
```

Selection SORT: complessità

```
function v = SelectionSort( v )

n = length(v);
for i = 1 : ( n - 1 )
    indiceMin = i;
    for j = (i + 1) : n
        if v( j ) < v( indiceMin )
            indiceMin = j;
        end
    end
    temp = v( indiceMin );
    v( indiceMin ) = v( i );
    v( i ) = temp;
end

end
```

'n-1' volte (circa n)

'n - i' volte

Complessità del Selection Sort

- Ciclo più esterno circa n volte, quello interno n – i
- Nel ciclo interno, al primo giro esterno si ha:

n - 1

- Al secondo giro, al totale si aggiunge un termine:

(n - 1) + (n - 2)

- In generale, proseguendo sino alla fine:

(n - 1) + (n - 2) + … +(n – n - 1) + (n – n) (ossia **n** volte **n-i**)

Applicando proprietà e risultati delle sommatorie (TESI, da dimostrare):

$$\sum_{i=1}^{n} n - i = \frac{n(n-1)}{2}$$

- Infatti, ricordiamo tre risultati generali:

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

- e

$$\sum_{i=1}^{n} c = \underbrace{c + c + c + \ldots . + c}_{n.volte} = c \cdot n$$

- Infine, ricordiamo la proprietà associativa:

$$\sum_{k=N}^{M} f(k) \pm \sum_{k=N}^{M} g(k) = \sum_{k=N}^{M} f(k) \pm g(k)$$

- Tramite la proprietà associativa scomponiamo n – i in due sommatorie e otteniamo (relazione A)

$$\sum_{i=1}^{n}(n - i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i$$

- Ricordando che

$$\sum_{i=1}^{n}(c) = c * n \quad \Rightarrow \quad \sum_{i=1}^{n} n = n^2$$

- E ricordando: $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n}(n - i) = n^2 - \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n}(n-i) = n^2 - \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} n - i = \frac{n(n-1)}{2}$$

Sommatorie

- https://it.wikipedia.org/wiki/Sommatoria

Algebra degli O grandi

Due cicli uno dentro l'altro:

Il blocco più esterno **n** operazioni (**n** passi base)
Il blocco più interno **n** operazioni (**n** passi base)

Applicando l'algebra degli O grandi:

$$O(g1(n) * g2(n)) = O(g1(n)) * O(g2(n))$$

Dunque la complessità è : n*n ossia n quadro

=stesso risultato ottenuto con i passi base

$$\sum_{i=1}^{n} n - i = \frac{n(n-1)}{2}$$

Insertion Sort

- E' di ordine 'n' se la sequenza è ordinata
- E' di ordine n^2 se completamente disordinata

Algoritmo	Caso migliore	Caso medio	Caso peggiore
SELECTIONSORT	n^2	–	n^2
INSERTIONSORT	n	–	n^2
BUBBLESORT	n	–	n^2
QUICKSORT	$n \log_2 n$	$n \log_2 n$	n^2
MERGESORT	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
HEAPSORT	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
COUNTINGSORT	n	n	n
BUCKETSORT	–	–	n

Definizioni, nomenclatura

- O(1): algoritmo che impiega sempre lo stesso tempo, indipendentemente dalla taglia;
- O(log(n)): algoritmo logaritmico;
- O(n): algoritmo lineare;
- O(n^k): algoritmo polinomiale;
- O(a^n): algoritmo esponenziale

Nell'insieme, si dovrebbero preferire algoritmi con un migliore (ossia più basso) limite asintotico: un algoritmo O(n^2) sarà preferibile ad uno O(n^3) anche se il coefficiente moltiplicativo dovesse essere maggiore.

Gli algoritmi polinomiali sono considerati trattabili

Trattabilità

- Algoritmi trattabili: problemi per cui esistono algoritmi la cui complessità sia di tipo polinomiale;
- Algoritmi intrattabili i problemi per cui un tale algoritmo non esiste (ma esiste di altra complessità).

In ascissa la taglia (10,20…), in ordinata la complessità, si assume che una operazione elementare = 1e-6 sec

f	10	20	40	60
n	$1e^{-5}$	$2e^{-5}$	$4e^{-5}$	$6e^{-5}$
n^3	$1e^{-3}$	$8e^{-3}$	$7e^{-2}$	$2e^{-1}$
n^5	$1e^{-1}$	3.2	1.7 min.	13 min.
2^n	$1e^{-3}$	1	13 giorni	36600 anni
3^n	$6e^{-2}$	1 ora	$4e^5$ anni	$1e^{13}$ anni

Trattabilità

In questa tabella indichiamo i miglioramenti ottenibili, in termini di Dimensioni delle istanze risolvibili, per diverse funzioni di complessità, al migliorare della tecnologia dei calcolatori. La colonna delle x è la dimensione dell'istanza risolvibile oggi in un minuto. *Il miglioramento della tecnologia non riduce significativamente il tempo di esecuzione di alcune importanti classi di algoritmi.*

f	Computer odierno	100 volte più veloce	10000 volte più veloce
n	x_1	$100x_1$	$10000x_1$
n^3	x_2	$4.6x_2$	$21.5x_2$
n^5	x_3	$2.5x_3$	$6.3x_3$
2^n	x_4	$x_4 + 6.6$	$x_4 + 13.2$
3^n	x_5	$x_5 + 4.2$	$x_5 + 8.4$

Trattabilità

N = numero dati gestiti in un lasso di tempo «t» (dipendente dalla complessità)

- Ad es.: 2^N operazioni gestite nel tempo t (x_4 = N).
- Con tecnologia 100 volte più veloce 2^N operazioni in t/100
- quindi 2^N *100 operazioni nel tempo t
- 100 è circa = $2^{6,64}$ …pertanto…
- 2^N *$2^{6,64}$ = $2^{N+6,64}$ operazioni nel tempo t ossia N + 6.6.

Se ne deduce:

- E’ più significativa l’evoluzione di un algoritmo rispetto all’evoluzione delle risorse elaborative

Esempio su ordinamento

Esempio Pratico: Ordinamento

Supponiamo di avere un array di 10^6 elementi:

- Con un algoritmo O(n^2) (Bubble Sort), il tempo di esecuzione è proporzionale a 10^12 operazioni.
- Su un processore da 3 GHz (3 miliardi di operazioni al secondo), ci vorrebbero circa **333 secondi**.
- Raddoppiando la velocità del processore (6 GHz), il tempo scende sa **166 secondi**.

Invece:

- Utilizzando un algoritmo O(n*log(n)) (Merge Sort), l'ordinamento richiederebbe 10^6 * 20 ≈ 2*10^7 operazioni.
- Lo stesso processore da 3 GHz potrebbe completare l'operazione in **0,0067 secondi**.

Trasformata di Fourier

- Vi ricordate il teorema del campionamento?
- Spettro di un segnale?
- L'algoritmo per il calcolo dello spettro di un segnale tempo discreto si basava sulla trasformata di Fourier (DFT):

$$F(j) \models \sum_{k=0}^{N-1} f(k) e^{-\frac{kj}{N} 2\pi i}, \qquad j = 0, \ldots, N-1.$$

La complessità di questo algoritmo è di ordine O(n^2). Nel 1965 Cooley e Tuckey scoprirono un nuovo algoritmo che, sotto alcune condizioni, faceva le stesse cose della DFT e lo chiamarono FFT = Fast Fourier Transform

Trasformata di Fourier : FFT

- L'algoritmo per il calcolo dello spettro del segnale, basato sulla FFT ha una complessità di ordine O(n*log(n)). Quindi:
  - Il primo algoritmo (DFT) riesce ad calcolare lo spettro di una sequenza di *n* numeri (campioni) con *n*^2 istruzioni
  - Il secondo (FFT) con *n* * log(*n*) istruzioni

DFT vs FFT

Supponiamo che l'esecuzione di un'istruzione avvenga in un μsec (10^{-6} sec)

Tempo di esecuzione:

	n=10	n=10000	$n=10^6$
n^2 op.	0,1 msec	100 sec (1,5 min)	10^6 sec (~12 gg.)
n*log n op.	23 μsec	92 msec	13,8 sec

Esempi ordine di grandezza

n	$O(n)$	$O(n\log(n))$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1.0000e+00	0.0000e+00	1.0000e+00	1.0000e+00	2.0000e+00
2	2.0000e+00	4.0000e+00	4.0000e+00	8.0000e+00	4.0000e+00
4	4.0000e+00	1.6000e+01	1.6000e+01	6.4000e+01	1.6000e+01
8	8.0000e+00	4.8000e+01	6.4000e+01	5.1200e+02	2.5600e+02
16	1.6000e+01	1.2800e+02	2.5600e+02	4.0960e+03	6.5536e+04
32	3.2000e+01	3.2000e+02	1.0240e+03	3.2768e+04	4.2950e+09
64	6.4000e+01	7.6800e+02	4.0960e+03	2.6214e+05	1.8447e+19
128	1.2800e+02	1.7920e+03	1.6384e+04	2.0972e+06	3.4028e+38
256	2.5600e+02	4.0960e+03	6.5536e+04	1.6777e+07	1.1579e+77
512	5.1200e+02	9.2160e+03	2.6214e+05	1.3422e+08	1.3408e+154
1024	1.0240e+03	2.0480e+04	1.0486e+06	1.0737e+09	Inf
2048	2.0480e+03	4.5056e+04	4.1943e+06	8.5899e+09	Inf
4096	4.0960e+03	9.8304e+04	1.6777e+07	6.8719e+10	Inf
8192	8.1920e+03	2.1299e+05	6.7109e+07	5.4976e+11	Inf
16384	1.6384e+04	4.5875e+05	2.6844e+08	4.3980e+12	Inf
32768	3.2768e+04	9.8304e+05	1.0737e+09	3.5184e+13	Inf
65536	6.5536e+04	2.0972e+06	4.2950e+09	2.8147e+14	Inf
131072	1.3107e+05	4.4564e+06	1.7180e+10	2.2518e+15	Inf
262144	2.6214e+05	9.4372e+06	6.8719e+10	1.8014e+16	Inf
524288	5.2429e+05	1.9923e+07	2.7488e+11	1.4412e+17	Inf
1048576	1.0486e+06	4.1943e+07	1.0995e+12	1.1529e+18	Inf

Esempi ordine di grandezza

Complessità	n=10	n=100	n=1000	n=10^6
$\sqrt{n}$	$3*10^{-6}$	10^{-5}	$3*10^{-5}$	10^{-3}
$n+5$	$15*10^{-6}$	10^{-4}	10^{-3}	1 sec
$2*n$	$2*10^{-5}$	$2*10^{-4}$	$2*10^{-3}$	2 sec
n^2	10^{-4}	10^{-2}	1 sec	10^6 (~12 gg.)
n^2+n	10^{-4}	10^{-2}	1 sec	10^6 (~12 gg.)
n^3	10^{-3}	1 sec	10^5 (~ 1 g)	10^{12} (~300 secoli)
2^n	10^{-3}	~ $4*10^{14}$ secoli	~ $3*10^{287}$ secoli	~ $3*10^{301016}$ secoli

FFT e Shannon

- Cose che non potrebbero esistere SENZA l'algoritmo FFT ed il Teorema del Campionamento
  - CD
  - JPEG
  - DVD
  - Digital TV
  - Cellulari
  - ABS, ESP, Common-rail etc. => in generale processi controllati da sistemi embedded in tempo reale
  - …..

- In taluni casi, per problemi di piccola taglia, si può verificare che un algoritmo di ordine superiore sia preferibile ad uno potenzialmente più efficiente: per esempio 5*n^3 è migliore di 100*n^2 se n < 20; inoltre alcuni algoritmi sono convenienti solo per ingressi di taglia elevatissima => inutilizzabili nella realtà;
- Se un programma verrà usato una o due volte il tempo speso per scriverlo diventa importante: se l'algoritmo più lento è più facile da codificare, sarà preferibile ad uno più veloce;
- In qualche caso l'algoritmo più veloce richiede troppo spazio;
- In qualche caso un algoritmo può essere migliore in media ma allo stesso tempo molto poco efficiente nel caso peggiore (e.g. quicksort);