Ingegneria Informatica

Algoritmi e strutture dati

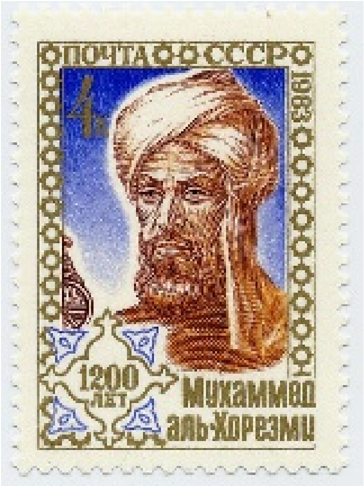
a.a. 2020-21

Prof.ssa Gigliola Vaglini

Concetti di base della complessità

Queste diapositive sono rielaborate a partire da quelle della Prof.ssa de Francesco

Origine del nome «algoritmo»



Francobollo commemorativo
Stampato in Unione sovietica
nel 1983 presunto1200° anno
dalla nascita

Pagina del testo originale di
Liber algebrae et almucabala

Muhammad ibn Musa al-Khwarizmi fu un matematico persiano dell'800 (libri scritti 813-830 circa).
Nel testo *al-Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa al-muqābala* raccolse materiali da culture differenti e li sistematizzò. Il libro fu tradotto per la prima volta parzialmente nel 1145 in latino, la parola algebra deriva dalla latinizzazione di *al-jabr,* che significa "completare", e si riferisce a una delle due operazioni usate per risolvere le equazioni di secondo grado.
L'altro testo fondamentale riguarda l'aritmentica e s'intitola *Algoritmi de numero Indorum* ("al-Khwārizmī sui numeri indiani") ed è sopravvissuto solo in una traduzione latina del 1126, mentre l'originale in arabo si è perso (il titolo era probabilmente *Kitāb al-Jamʿ wa al-tafrīq bi-ḥisāb al-Hind)*. E' la prima opera completa sul sistema di numerazione indiano e si deve ad esso la diffusione in occidente della notazione posizionale e dello 0.

Il nome algoritmo con il significato attuale viene usato dal XIX secolo

Prime caratterizzazioni di algoritmo e computer

Alonso Church: lambda-calcolo (1936)

sistema formale per analizzare il calcolo delle funzioni definito per mezzo di riscritture che portano a semplificazioni dei termini

Alan Turing : Macchina di Turing (1936)

modello matematico capace di simulare qualunque procedimento algoritmico

Concetto di algoritmo

- Input/Output
- insieme finito di istruzioni teso a risolvere un problema
- ogni istruzione deve essere ben definita ed eseguibile in un tempo finito da un agente di calcolo
- E’ possibile utilizzare una memoria per i risultati intermedi

Primi algoritmi nella storia

Algoritmi aritmetici

Babilonesi (circa 2500 a.c.)

Egizi (circa 1500 a.c.)

Greci: **algoritmo di Euclide (300 a.c.)**

setaccio di Eratostene (sieve, crivello) (I secolo d.c.)

Algoritmo di Euclide

Trovare il Massimo Comun Divisore fra due numeri interi non negativi

P1: Il MCD fra due numeri interi positivi è uguale al MCD fra il più piccolo e la differenza fra i due

Se m ed n sono divisibili per x, allora anche (m-n), con m>n, è divisibile per x, ovvero la differenza conserva i fattori comuni di m e n

$m=xk \quad n=xh$

$m-n= xk-xh=x(k-h)$

Utilizzare la proprietà

L'algoritmo è basato sulle sottrazioni successive, il risultato deve essere sempre positivo e quindi devo sempre sottrarre il numero più piccolo dal più grande

Passo

se $m > n$ allora $m-n$ altrimenti $n-m$

Il metodo converge perché diminuisco sempre il numero più grande e conservo il più piccolo

Halt

Quando m ed n sono uguali (ovvero la differenza tra m e n è uguale a 0) e quindi so dare direttamente la risposta al problema

Esempio di calcolo

MCD(30, 21)

```
int MCD(int x, int y) {
    while (x!=y)
    if (x < y) y=y-x; else x=x-y;
    return x;
}
```

x= 30, y=21

x= 9, y=21

x = 9, y=12

x =9, y= 3

x =6, y=3

x= 3, y=3

Altra proprietà

P2: Il MCD fra due numeri m e n (m>n) è uguale al MCD fra n e r, resto della divisione intera fra m e n

Ogni numero intero che divide m e n divide anche r perché divide la differenza

Passo dell'algoritmo
Se m>n e $r \neq 0$ m div n

Il metodo converge perché il resto è sempre più piccolo del più piccolo tra m e n e conservo il più piccolo tra m e n.

Halt
Se r=0, il procedimento si ferma e il MCD è il resto precedente

Algoritmo di Euclide con il resto della divisione

MCD(30, 21)

```
int MCD(int x, int y)
{
   while (y != 0)
 { int k=x;
   x=y;
   y=k%y; }
   return x;
 }
```

x= 30, y=21

x= 21, y=9

x = 9, y=3

x = 3, y=0

Confronti

I due algoritmi sono equivalenti

Hanno tempi di esecuzione differenti

Usano una quantità di memoria differente

Richiedono macchine di costo differente

Numeri primi

Trovare tutti i numeri primi fino a n

Algoritmo inefficiente: dividere ogni numero minore o uguale a n per tutti i suoi predecessori

Algoritmo poco più efficiente: dividere ogni numero n per tutti i suoi predecessori da 2 a radice quadrata di n.

Algoritmo molto più efficiente

Setaccio di Eratostene

1. Elencare tutti i numeri maggiori di 1 fino a n
2. Partendo dal numero primo 2, cancellare dall'elenco tutti i multipli di 2
3. Ripetere il procedimento con i numeri seguenti non ancora cancellati

Setaccio di Eratostene

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Setaccio di Eratostene

Cancello i multipli di 2

	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	~~100~~

Setaccio di Eratostene

Cancello i multipli di 3 a partire da 9

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	~~100~~

Setaccio di Eratostene

Cancello i multipli di 5 a partire da 25

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	~~100~~

Setaccio di Eratostene

Cancello i multipli di 7 a partire da 49

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	36	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	~~100~~

Stop perché 11*11 > 100

Setaccio di Eratostene

```
void setaccio (int n){
bool primi[n]; primi[0] = primi[1] = false;
for (int i = 2; i < n; i++) primi[i] = true;  //inizializza
int i = 1;
for (i++; i * i < n; i++){ //scorri i numeri a partire da 2

   while (!primi[i]) i++; // cerca il prossimo primo
   for (int k=i * i; k< n; k+= i) primi[k]= false; // scorri i numeri
successivi a partire da i*i* e cancella multipli di i
   }
  for(int j = 2; j<n; j++)
      if (primi[j]) cout <<j<< endl; //stampa tutti primi
}
```

* i numeri non cancellati precedenti i*i sono primi

Complessità degli algoritmi (programmi)

Si dice complessità di un algoritmo

il risultato di una funzione che associa alla dimensione del problema il costo (sempre positivo) della sua risoluzione

La dimensione del problema: dipende dai dati

Il costo della soluzione può essere valutato in tempo, spazio (memoria), o altri parametri rilevanti per il problema

Per confrontare due algoritmi si confrontano le relative complessità

Calcolo della complessità dei programmi

$T_P(n)$ = Complessità in tempo di esecuzione del programma P al variare di n

```
int max(int a[], int n)
{
  int m=a[0];
  for (int i=1; i < n;i++)
     if (m < a [ i ]) m = a[i];
  return m;
}
```

Considerando il tempo per eseguire un'istruzione costante e uguale a 1:

$$T_{max}(n) = 4n$$

Complessità dei programmi

E' necessario però trovare un metodo di calcolo della complessità che misuri l'efficienza come proprietà dell'algoritmo, cioè astragga

- **dal computer su cui l'algoritmo è eseguito**
- **dal linguaggio in cui l'algoritmo è scritto**

L'efficienza deve essere misurata indipendentemente anche dalla specifica dimensioni dei dati

Piuttosto la complessità deve essere analizzata nel suo comportamento asintotico ovvero al crescere della dimensione

Esempio

$T_P(n) = 2n^2$ $T_Q(n) = 100n$ $T_R(n) = 5n$

Per n >= 50, $T_Q(n) <= T_P(n)$

$\Downarrow$

$T_Q(n)$ ha complessità minore o uguale a $T_P(n)$
ma non vale il contrario (vale solo per un numero finito di valori di n)

Per n >= 3, $T_R(n) <= T_P(n)$

$\Downarrow$

$T_R(n)$ ha complessità minore o uguale a $T_P(n)$
ma non vale il contrario

Esempio (cont)

$T_P(n) = 2n^2$ $T_Q(n) = 100n$ $T_R(n) = 5n$

Per ogni n, $T_R(n) \le T_Q(n)$

$T_R(n)$ ha complessità minore o uguale a $T_Q(n)$

Per ogni n, $T_Q(n) \le 20T_R(n)$

$T_Q(n)$ ha complessità minore o uguale a $T_R(n)$

Posso trovare una costante positiva da moltiplicare per $T_R(n)$

$T_Q(n)$ e $T_R(n)$ hanno la stessa complessità

Notazione O grande (limite asintotico superiore)

f(n) è di ordine O(g(n)) se esistono

un intero n_0 ed una costante $c>0$ tali che

per ogni $n \geq n_0$: $f(n) \leq c\ g(n)$

Notazioni

f(n) è di ordine O(g(n))

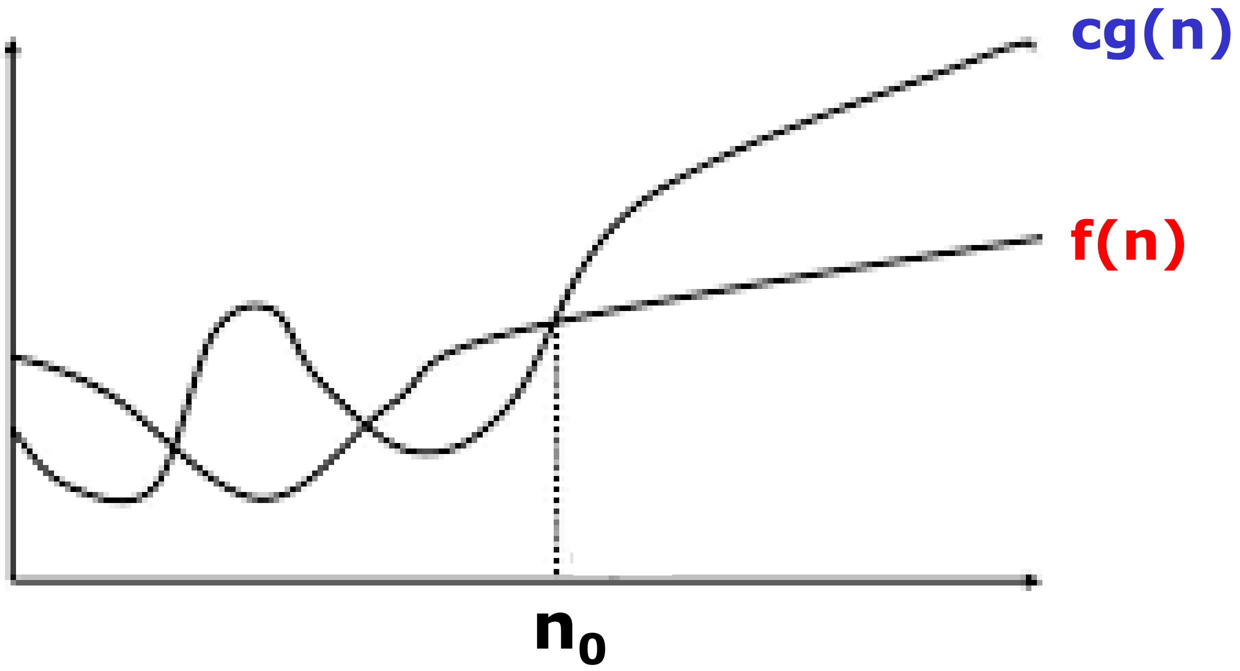
f(n) è O(g(n))

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

f(n) = O(g(n)) notazione ambigua

Notazione O grande

f(n) è O(g(n))



$T_Q(n)$ è $O(T_P(n))$ [n0=50, c=1] oppure [n0=1, c= 50]

$T_R(n)$ è $O(T_P(n))$ [n_0=3, c=1]

$T_R(n)$ è $O(T_Q(n))$ [n_0=1, c=1]

$T_Q(n)$ è $O(T_R(n))$ [n_0=1, c=20]

$T_P(n) = 2n^2$

$T_Q(n) = 100n$

$T_R(n) = 5n$

$T_P(n)$ non è $O(T_Q(n))$ anche se moltiplico $T_Q(n)$ per una costante prima o poi $T_P(n)$ la supera

$T_P(n)$ non è $O(T_R(n))$

Notazioni

f(n) è di ordine O(g(n))

f(n) è O(g(n))

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

Una funzione f(n)=expr si indica soltanto con expr

f(n)= 3-n ➔ **3-n**

f(n)=100n è O(g(n)=5n) ➔ **100n è O(5n)**

Esempi

$T_{max}(n) = 4n \in$ $O(n)$ $[\ n_0=1,\ c=4\ ]$

$T_{max}(n) = 4n \in$ $O(n^2)$ $[\ n_0=4,\ c=1\ ]$

$T_Q(n),\ T_R(n) \in$ $O(n)$

$2^{n+10} \in$ $O(2^n)$ $[\ n_0=1,\ c=2^{10}\ ]$

$n^2 \in$ $O(1/100\ n^2)$ $[\ n_0=1,\ c=100\ ]$

$n^2 \in$ è $O(2^n)$ $[\ n_0=4,\ c=1\ ]$

O(f(n)) = insieme delle funzioni

di ordine O(f(n))

$O(n) = \{ \text{costante}, n, 4n, 300n, 100 + n, .. \}$

$O(n^2) = O(n) \cup \{ n^2, 300n^2, n + n^2, \ldots \}$

Regole

REGOLA DEI FATTORI COSTANTI

Per ogni costante positiva k, O(f(n)) = O(kf(n))

REGOLA DELLA SOMMA

Se f(n) è O(g(n)), allora f(n)+g(n) è O(g(n))

REGOLA DEL PRODOTTO

Se f(n) è O(f1(n)) e g(n) è O(g1(n)), allora f(n)g(n) è O(f1(n)g1(n)).

Regole (cont)

- Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora

  $f(n)$ è $O(h(n))$

- per ogni costante k , k è $O(1)$

- per $m <= p$, n^m è $O(n^p)$

- Un polinomio di grado m è $O(n^m)$

Esempi

- $2n + 3n + 2$ è $O(n)$
- $(n+1)^2$ è $O(n^2)$
- $2n + 10 n^2$ è $O(n^2)$

Esempio

$$f(n) = \begin{cases} n^3 & \text{se } n \text{ è pari} \\ n^2 & \text{se } n \text{ è dispari} \end{cases} \qquad g(n) = \begin{cases} n^2 & \text{se } n \text{ è primo} \\ n^3 & \text{se } n \text{ è composto} \end{cases}$$

$f(n)$ è $O(g(n))$ $n0=3$, $c=1$

non vale il contrario: esistono infiniti numeri composti dispari e altrettanti pari

Funzioni incommensurabili

$$f(n) = \begin{cases} n & \text{se } n \text{ è pari} \\ n^2 & \text{se } n \text{ è dispari} \end{cases}$$

$$g(n) = \begin{cases} n^2 & \text{se } n \text{ è pari} \\ n & \text{se } n \text{ è dispari} \end{cases}$$

Classi di Complessità

O(1) **costante**

O(logn) **logaritmica**

Nel caso di complessità logaritmica, non si specifica la base del logaritmo, ossia, per ogni a e b, $O(\log_a n) = O(\log_b n)$. Infatti $\log_a n = (\log_b n)(\log_a b)$ e $\log_a b$ è una costante, quindi $\log_a n \in O(\log_b n)$.

O(n) **lineare**

Le funzioni con complessità minore di O(n) si dicono sottolineari (per esempio, oltre alle costanti, radice(n) è sottolineare) mentre quelle con complessità maggiore si dicono sopralineari.

Classi di complessità (cont)

$O(n \log n)$ **nlogn**

$O(n^2)$ **quadratica**

$O(n^3)$ **cubica**

..

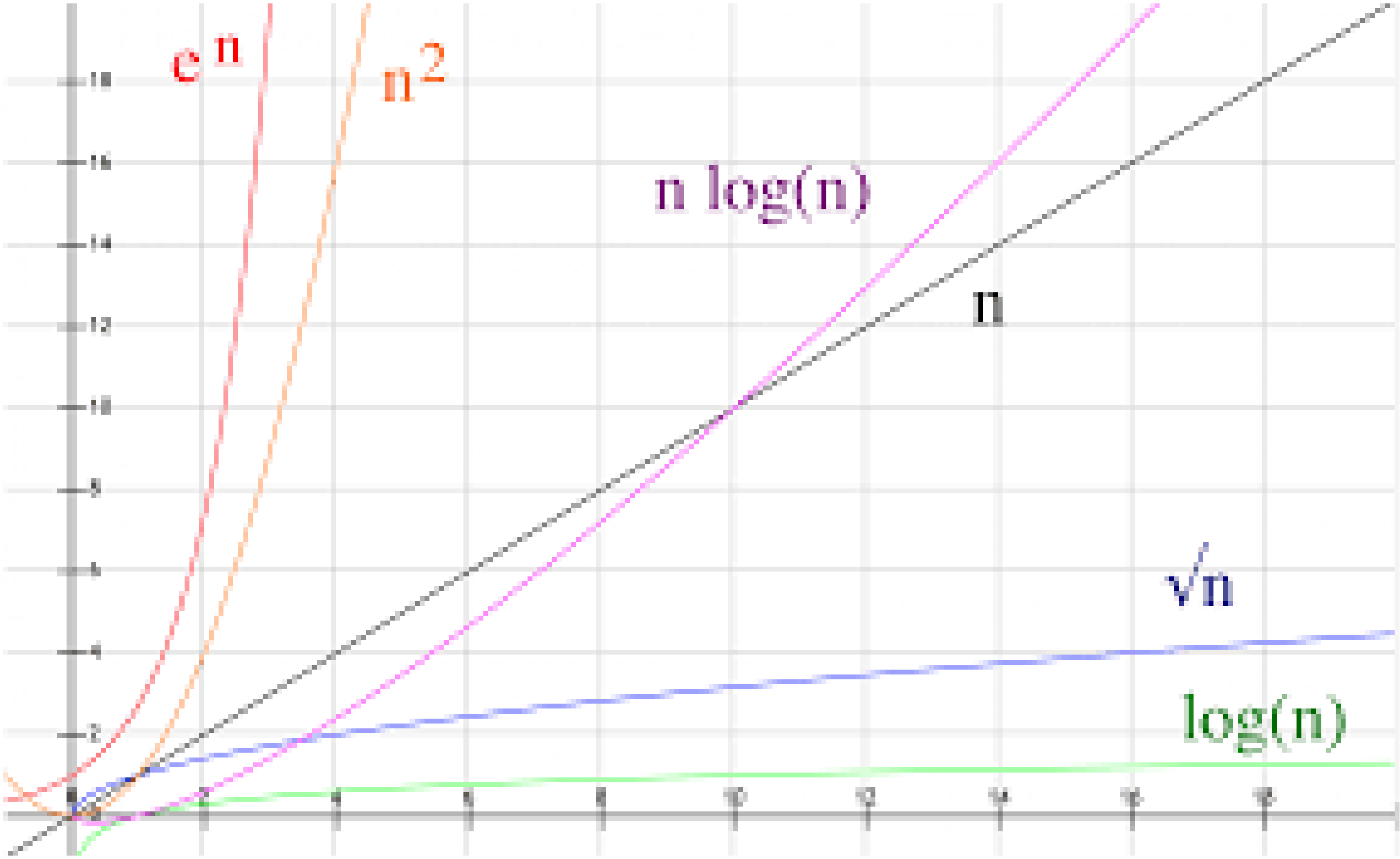
$O(n^p)$ **polinomiale**

Ogni classe $O(n^k)$, per $k \geq 0$, è comunque detta di complessità polinomiale.

$O(2^n)$ **esponenziale**

$O(n^n)$ **esponenziale**

Classi di Complessità



Teorema

per ogni k e per ogni $a > 1$ $\quad n^k \in O(a^n)$,

Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale

Crescita esponenziale

$2^{64}-1$ chicchi : 18.446.744.073.709.551.615 chicchi, superiore ai raccolti di grano di tutto il mondo

Setaccio di Eratostene

```
void setaccio (int n){
bool primi[n]; primi[0] = primi[1] = false;
for (int i = 2; i < n; i++) primi[i] = true;  //inizializza
int i = 1;
for (i++; i * i < n; i++){ //scorri i numeri successivi a partire da 2
 while (!primi[i]) i++; // cerca il prossimo primo
   for (int k=i * i; k< n; k+= i) primi[k]= false; // scorri i numeri
successivi a partire da i*i* e cancella multipli di i
   }
  for(int j = 2; j<n; j++)
      if (primi[j]) cout <<j<< endl; //stampa tutti primi
}
```

* i numeri non cancellati precedenti i*i sono primi

Esercizio 1

Calcolare la complessità in funzione di n>0 del seguente frammento di programma:

```
        for (int j=1; j<=f(n);j++) a+=n
```

con la seguente definizione di f:

```
  int f (int n){
          int a=0;
          for (int j=1; j<=n;j++) a+=n;
          return a;
}
```

Esercizio 2

Dire, per ogni coppia di funzioni fra quelle definite sotto, se una è O dell'altra oppure no.

$$f(n) = \begin{cases} 3n^3 + 3n & \text{se } n \text{ è primo} \\ n & \text{altrimenti} \end{cases}$$

$$g(n) = \begin{cases} 4n^3 & \text{se l'ultima cifra di } n \text{ è } 0 \text{ o } 5 \\ n^3 & \text{altrimenti} \end{cases}$$

$$h(n) = \begin{cases} n^2 & \text{se } n \text{ è divisore di } 50 \\ n^3 & \text{altrimenti} \end{cases}$$

Esercizio 3

Calcolare la complessità in funzione di n>=0 della seguente funzione:

```
int g (int n)
{ int a=n;
   if (n<=500) for (int i=1; i<=n; i++)
                          for (int j=1; j<=n;j++) a+=n;
   else for (int i=1; i<=n; i++) a+=n;
   return a;
}
```

Esercizio 4

Dato il seguente frammento di programma:

```
i=n;
while (i>=1) { for (int j=1; j<=n;j++) a++; i=E;}
```

calcolare la complessità in funzione di $n>0$ nei casi

a) **E=i-1**

b) **E=i-n**

c) **E=i/2.**