Ingegneria degli Algoritmi

Alessandro Pellegrini
pellegrini@diag.uniroma1.it

Informazioni Generali

- Informazioni sul docente
  - email: pellegrini@diag.uniroma1.it
  - URL: http://www.diag.uniroma1.it/~pellegrini/
- Testo consigliato:
  - T. Cormen, C. Leiserson, L. Rivest, R. Stein. *Introduction to algorithms.* 3a ed. The MIT Press, 2009, Cambridge
- Lezioni:
  - Lunedì, 14.00-15.30 (Aula C1)
  - Mercoledì, 9.30-11.00 (Aula B2)
  - ~~Venerdì, 14.00-15.30 (Aula C1)~~
- Ricevimento:
  - Al termine di ogni lezione (ma avremo poco tempo)
  - Secondo il calendario pubblicato sul sito

Obiettivi del Corso

- Prendere confidenza con la progettazione e l'analisi di algoritmi
- Capire come si misura l'efficienza degli algoritmi e delle strutture dati
- Imparare a scegliere quale algoritmo è più conveniente utilizzare per risolvere problemi del mondo reale
- Implementare algoritmi e strutture dati in Python

I would rather have today's algorithms on yesterday's computers than vice versa

—Philippe Toint, math prof, Belgium

Informazioni sull'esame

- L'esame consiste in una *prova scritta*
- La prova scritta conterrà sia domande *teoriche* che domande *pratiche*
- ***Challenge Algoritmico***:
  - Durante il corso, verranno pubblicati tre problemi da risolvere
  - Gli studenti possono consegnare (entro scadenze prestabilite) dei programmi che risolvono questi problemi
  - I programmi corretti verranno organizzati in una graduatoria, in funzione della loro efficienza
  - I primi tre classificati riceveranno 2 punti in più che verranno sommati al voto finale
  - Si può totalizzare un massimo di 4 punti con la Challenge
  - I punti scadono al termine dell'anno accademico

Informazioni sull'esame

- Durante gli scritti
  - È vietato comunicare in qualunque modo, per qualsiasi motivo
  - Chi viene sorpreso a parlare, viene invitato a lasciare l'aula ed a ripresentarsi all'appello successivo
  - Questa regola vale per tutte le persone coinvolte nella comunicazione
  - Se avete bisogno di qualsiasi cosa, chiedete al docente
- Dopo gli scritti
  - Il compito potrà essere annullato anche in caso di manifesta copiatura scoperta durante la correzione degli scritti
  - L'annullamento riguarderà sia il *copiatore* sia il *copiato*

Argomenti del corso

- Tecniche algoritmiche
- Analisi della complessità
- Il problema dell'ordinamento
- Strutture dati di base: vettori, liste, code, pile
- Strutture dati avanzate: alberi, mucchi, insiemi
- Tecniche di hashing
- Grafi ed algoritmi sui grafi

Introduzione

Definizioni di base

- **Struttura dati**: un'organizzazione sistematica dei dati e del loro accesso, che ne facilita la manipolazione
- **Algoritmo**: procedura suddivisa in passi elementari che, eseguiti in sequenza, consentono di svolgere un compito in tempo finito

Muḥammad ibn Mūsā al-Khwārizmī

al-Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa al-muqābala

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Definizioni di base

- Un algoritmo è uno *strumento* per risolvere un **problema computazionale** ben definito
  - Dati un dominio di input e un dominio di output, un *problema computazionale* è rappresentato dalla *relazione matematica* che associa un elemento del dominio di output ad ogni elemento del dominio di input
- Un algoritmo è detto **corretto** se, per *ogni* istanza di input, l'algoritmo termina con l'output corretto
- Un algoritmo è detto **non corretto** se esiste almeno un'istanza di input per cui l'algoritmo non fornisce l'output corretto o non termina.

Algoritmi nella storia

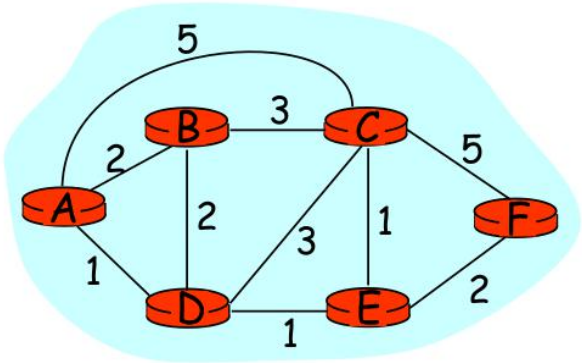
- Papiro di Rhind o di Ahmes (1850 a.C.): algoritmo del contadino per la moltiplicazione
- Algoritmi numerici studiati da matematici babilonesi ed indiani
- Algoritmi studiati da matematici greci (2000 anni fa):
  - Algoritmo di Euclide per il MCD
  - Algoritmi geometrici (calcolo di tangenti, sezioni di angoli, …)

Algoritmi nel mondo moderno

- Gli algoritmi intervengono (in maniera più o meno nascosta) in un numero molto grande di aspetti
  - Internet e connessione tra dispositivi
  - Data Base Management Systems
  - Motori di ricerca
  - Navigatori
  - Profilazione (suggerimenti di prodotti da acquistare o film da vedere)

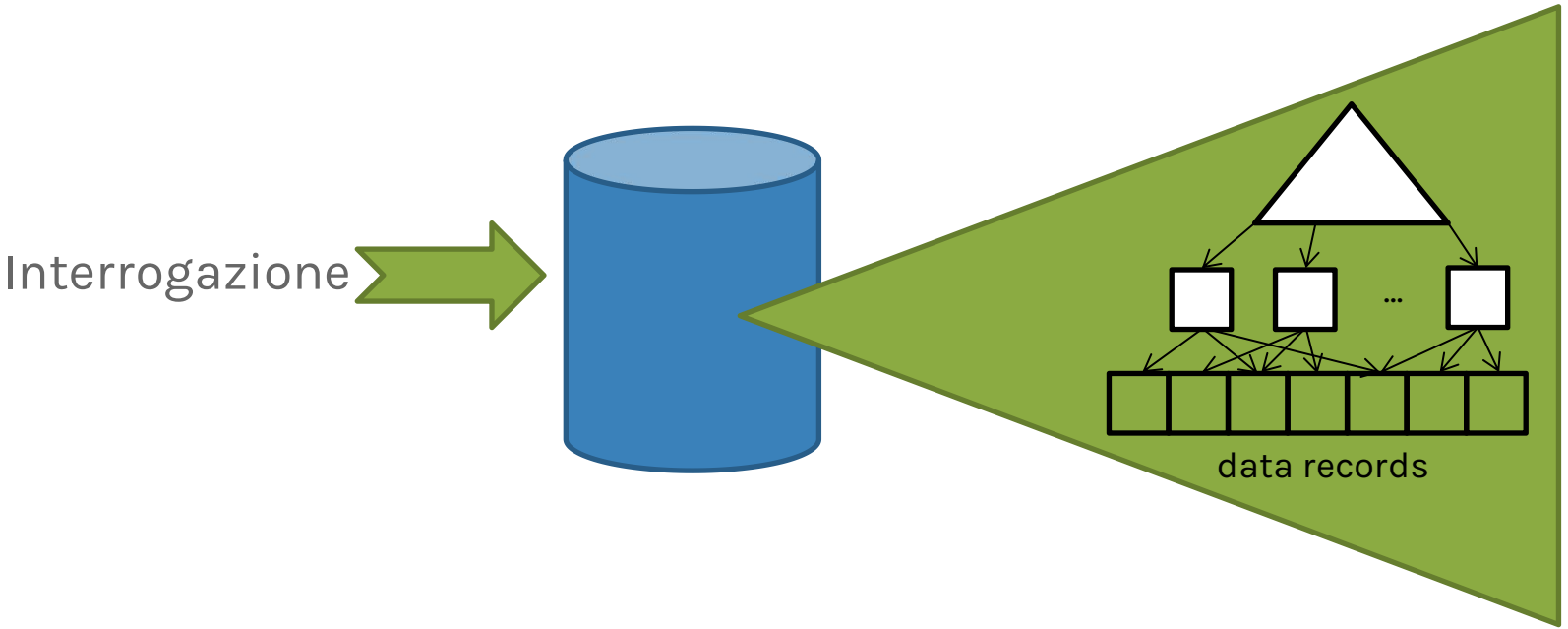
Esempio: Routing in Internet

- *Protocollo di Routing:*
  - Obiettivo: *determinare un buon cammino dalla sorgente alla destinazione*
- Gli archi descrivono le connessioni fisiche
- I nodi rappresentano i router
- Cammino “buono”: tipicamente a costo minimo.



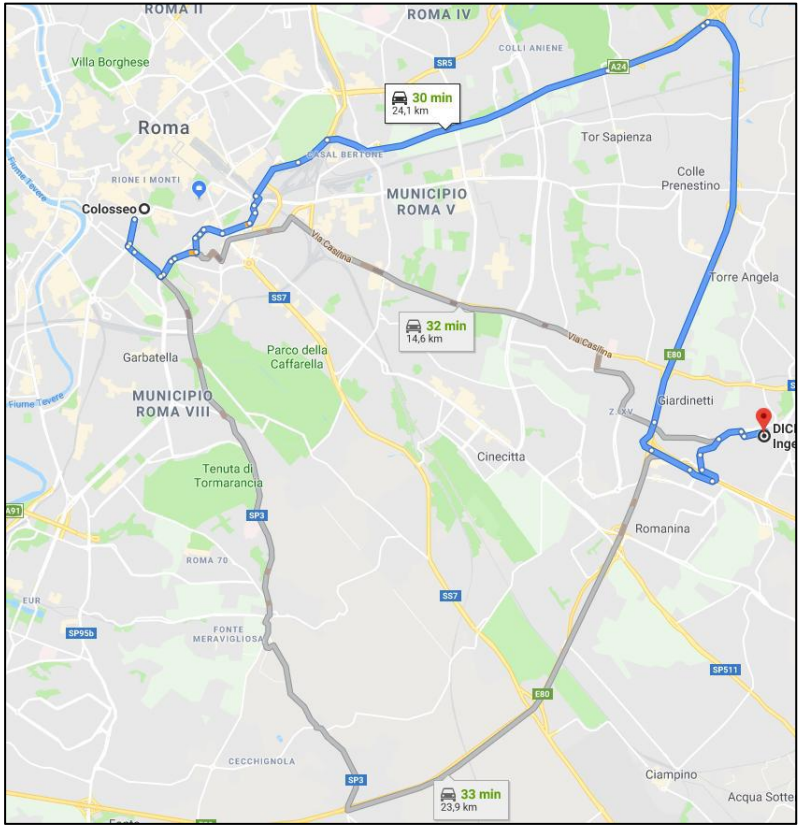
Esempio: Accesso a base di dati

- *Interrogazione*:
  - Obiettivo: *rispondere il più rapidamente possibile*



Esempio: Navigatori

- Identificazione del “miglior” percorso da seguire:
  - distanza minore
  - tempo di percorrenza minore
  - evitare aree o strade specifiche (es, percorsi senza pedaggio)



Colloquio di lavoro

Problema: sottovettore di somma massimale

- **Input**: un vettore di interi *A[1…n]*
- **Output**: il sottovettore *A[i…j]* di somma massimale, ovvero il sottovettore la cui somma degli elementi $\sum_{k=i}^{j} A[k]$ è più grande o uguale alla somma degli elementi di qualunque altro sottovettore.

Colloquio di lavoro

Problema: sottovettore di somma massimale

- **Input**: un vettore di interi *A[1…n]*
- **Output**: il sottovettore *A[i…j]* di somma massimale, ovvero il sottovettore la cui somma degli elementi $\sum_{k=i}^{j} A[k]$ è più grande o uguale alla somma degli elementi di qualunque altro sottovettore.

- Il problema è scritto bene?

Colloquio di lavoro

Problema: sottovettore di somma massimale

- **Input**: un vettore di interi *A[1…n]*
- **Output**: il sottovettore *A[i…j]* di somma massimale, ovvero il sottovettore la cui somma degli elementi $\sum_{k=i}^{j} A[k]$ è più grande o uguale alla somma degli elementi di qualunque altro sottovettore.

- Il problema è scritto bene?

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Colloquio di lavoro

Problema: sottovettore di somma massimale

- **Input**: un vettore di interi *A[1…n]*
- **Output**: il sottovettore *A[i…j]* di somma massimale, ovvero il sottovettore la cui somma degli elementi $\sum_{k=i}^{j} A[k]$ è più grande o uguale alla somma degli elementi di qualunque altro sottovettore.

- Il problema è scritto bene?

1	3	4	-8	2	3	-1	3	4	-3	10	-3	2

Colloquio di lavoro

Problema: sottovettore di somma massimale

- **Input**: un vettore di interi *A[1…n]*
- **Output**: il sottovettore *A[i…j]* di somma massimale, ovvero il sottovettore la cui somma degli elementi $\sum_{k=i}^{j} A[k]$ è più grande o uguale alla somma degli elementi di qualunque altro sottovettore.

- Il problema è scritto bene?
- Riuscite a risolverlo?
- Riuscite a risolverlo in maniera efficiente?

Soluzione naïf al problema: $O(n^3)$

- Cicla su tutte le coppie *(i,j)* tali che $i \leq j$:
  - calcola la somma dei valori compresi *tra i e* j
  - aggiorna maxSoFar con il massimo fra la somma appena calcolata ed il massimo trovato fin'ora

```
def maxsum1(A):
    maxSoFar = 0; # Maximum found so far
    for i in range(0, len(A)):
        for j in range(i, len(A)):
            maxSoFar = max(maxSoFar, sum(A[i:j+1]))
    return maxSoFar;
```

Prima ottimizzazione: $O(n^2)$

- *Intuizione*: se ho calcolato la somma *s* dei valori in *A[i…j]*, la somma dei valori in *A[i…j+1]* è pari ad *s + A[j+1]*.

```
def maxsum2(A):
    maxSoFar = 0 # Maximum found so far
    for i in range(0, len(A)):
        sum = 0 # Accumulator
        for j in range(i, len(A)):
            sum = sum + A[j]
            maxSoFar = max(maxSoFar, sum)
    return maxSoFar
```

Seconda ottimizzazione: *O(n logn)*

- Divide et impera:
  - dividiamo il vettore nelle metà di destra e sinistra, in due parti più o meno uguali
  - `maxL` è la somma massimale nella parte sinistra
  - `maxR` è la somma massimale nella parte destra
  - Restituisce il massimo dei due valori
- È corretta questa implementazione?

`maxL` `maxR`

Seconda ottimizzazione: *O(n logn)*

- Divide et impera:
  - dividiamo il vettore nelle metà di destra e sinistra, in due parti più o meno uguali
  - `maxL` è la somma massimale nella parte sinistra
  - `maxR` è la somma massimale nella parte destra
  - `maxLL+maxRR` è il valore della sottolista massimale "al centro"
  - Restituisce il massimo dei tre valori

Seconda ottimizzazione: *O(n logn)*

```
def maxsum_rec(A,i,j):
  if (i==j):
    return max(0, A[i])
  m = (i+j)//2
  maxLL = 0 # Maximal subvector on the left ending in m
  sum = 0
  for k in range(m, i-1,-1):
    sum = sum + A[k]
    maxLL = max(maxLL, sum);
  maxRR = 0 # Maximal subvector on the right starting in m+1
  sum = 0
  for k in range(m+1,j+1):
    sum = sum + A[k]
    maxRR = max(maxRR, sum);
  maxL = maxsum_rec(A, i, m) # Maximal subvector on the left
  maxR = maxsum_rec(A, m+1, j) # Maximal subvector on the right
  return max(maxL, maxR, maxLL + maxRR)

def maxsum3(A):
  return maxsum_rec(A,0,len(A)-1)
```

Terza ottimizzazione: *O(n)*

Programmazione dinamica

- Sia *maxHere[i]* il valore del sottovettore di somma massima che termina in posizione *A[i]*

$$maxHere[i] = \begin{cases} 0 & i < 0 \\ \max(maxHere[i-1] + A[i], 0) & i \geq 0 \end{cases}$$

- Viene tenuta traccia di quanto calcolato fino ad un certo punto di esecuzione dell'algoritmo

Terza ottimizzazione: *O(n)*

```
def maxsum4(A):
  maxSoFar = 0 # Maximum found so far
  maxHere = 0 # Maximum slice ending at the current pos
  start = end = 0 # Start, end of the maximal slice found so far
  last = 0 # Beginning of the maximal slice ending here
  for i in range(0, len(A)):
    maxHere = maxHere + A[i]
    if maxHere <= 0:
      maxHere = 0
      last = i+1
    if maxHere > maxSoFar:
      maxSoFar = maxHere
      start, end = last, i
  return (start, end)
```

Terza ottimizzazione: *O(n)*

A	1	3	4	-8	2	3	-1	3	4	-3	10	-3	2
maxHere	1	4	8	0	2	5	4	7	11	8	18	15	17
maxSoFar	1	4	8	8	8	8	8	8	11	11	18	18	18
last	0	0	0	4	4	4	4	4	4	4	4	4	4
start	0	0	0	0	0	0	0	0	4	4	4	4	4
end	0	1	2	2	2	2	2	2	8	8	10	10	10

- La colonna associata ad ogni elemento del vettore contiene il valore delle variabili dopo l'esecuzione del ciclo per quell'elemento

Esempi di problemi computazionali

- Minimo:
  - Il minimo di un insieme S è l'elemento di S che è minore di o uguale ad ogni altro elemento di S.

$$\min(s) = a \Leftrightarrow \exists a \in S : \forall b \in S : a \leq b$$

- Ricerca:
  - Sia $S = s_1, s_2, \ldots, s_n$ una sequenza di dati ordinati e distinti, ovverosia $s_1 \leq s_2 \leq \ldots \leq s_n$. Eseguire una ricerca della posizione di un dato $v \in S$ consiste nel restituire un indice $1 \leq i \leq n$ se v è presente nella posizione i, oppure 0 se v non è presente.

$$lookup(S, v) = \begin{cases} i & \exists i \in \{1, \ldots, n\} : S_i = v \\ 0 & \text{altrimenti} \end{cases}$$

Esempi di algoritmi

- Minimo:
  - Per trovare il minimo di un insieme, confronta ogni elemento con tutti gli altri; l'elemento che è minore di tutti è il minimo
- Ricerca:
  - Per trovare un valore *v* nella sequenza *S*, confronta *v* con tutti gli elementi di *S*, in sequenza, e restituisci la posizione corrispondente; restituisci 0 se nessuno degli elementi corrisponde.

Come descrivere e studiare un algoritmo

- Le descrizioni precedenti hanno un problema di **descrizione**
  - Il linguaggio naturale tende ad essere fortemente *impreciso* ed *ambiguo*
- Gli algoritmi possono essere rappresentati in maniera indipendente dal linguaggio che verrà poi utilizzato per implementare l'algoritmo stesso: lo **pseudocodice**
- Questa rappresentazione deve essere il più possibile *formale*
- Lo pseudocodice consente anche di effettuare un'analisi degli algoritmi *indipendente dall'architettura di calcolo*
  - L'architettura di riferimento può però avere un grande impatto sulle implementazioni che sono possibili e sulla loro efficienza

Esempi di pseudocodice

- Calcolo del massimo in un vettore di *n* elementi:

```
arrayMax(A, n):
    currentMax ← A[1]
    for i ← 1 to n:
        if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
```

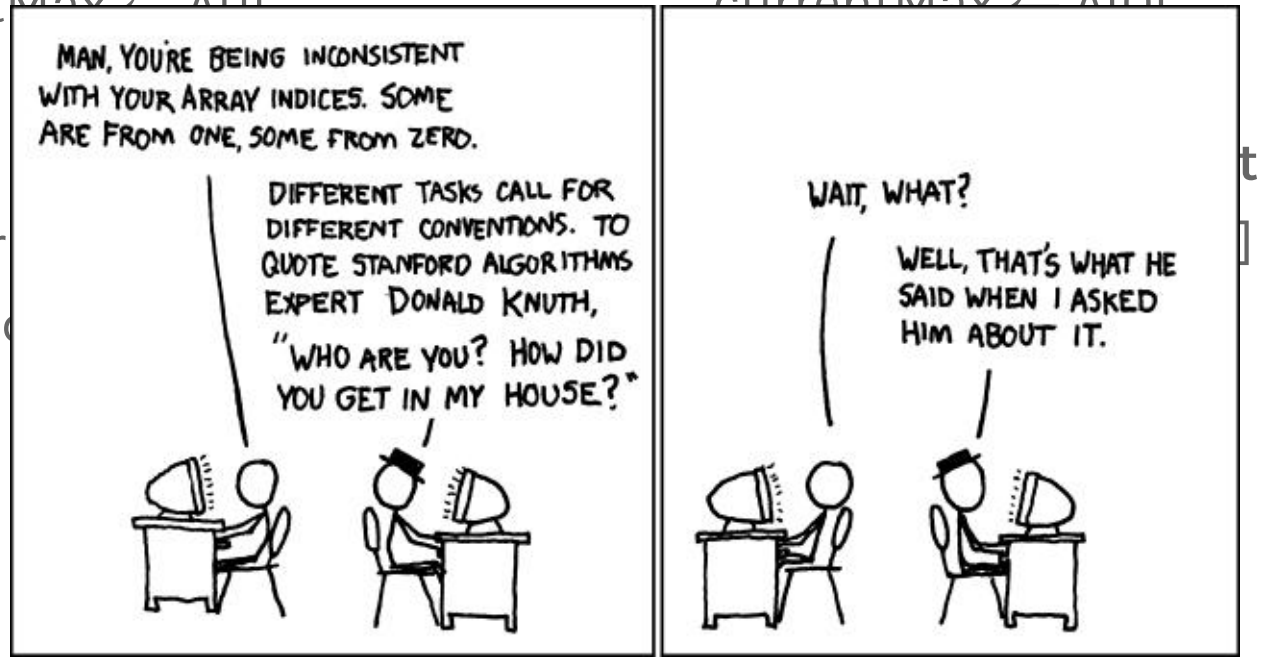
```
arrayMax(A, n):
    currentMax ← A[0]
    for i ← 0 to n - 1:
        if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
```

Esempi di pseudocodice

- Calcolo del massimo in un vettore di *n* elementi:

```
arrayMax(A, n):
    currentMax ← A[1]
    for i ←
        if A[i]
            cur
    return
```

```
arrayMax(A, n):
    currentMax ← A[0]
                                  then
                                  ]
```



Qualità degli algoritmi

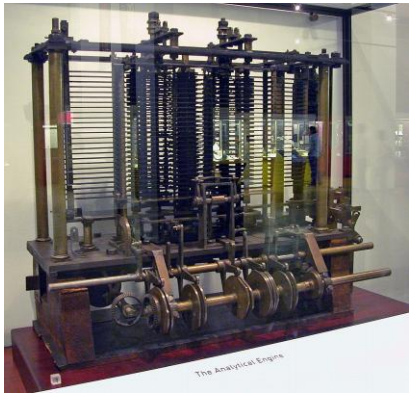
- **Efficienza**:
  - Tempo di esecuzione
  - Spazio (quantità di memoria)
- I due aspetti sono interdipendenti

- Uno stesso problema può essere risolto mediante più algoritmi
- L'efficienza di algoritmi differenti può essere incredibilmente differente
  - Queste differenze possono essere più significative delle differenze legate a hardware o tecnologie software usati

Come misurare l'efficienza

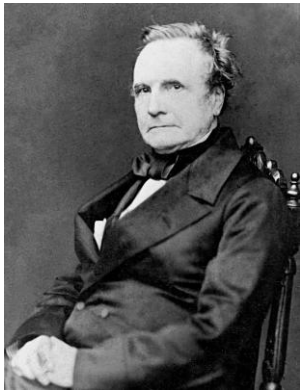
As soon as an Analytical Engine Exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise: "*By what course of calulation can these results be arrived at by the machine in the shortest time?*"

—Passages from the Life of a Philosopher, *Charles Babbage, 1864*



La macchina analitica

Schede perforate



Charles Babbage, 1860

Come misurare l'efficienza

- **Complessità di un algoritmo**
  - Analisi delle *risorse* impiegate da un algoritmo per risolvere un problema, in funzione della *dimensione* e della *tipologia* dell'input
- **Risorse**
  - *Tempo*: tempo impiegato per completare l'algoritmo (in che unità lo misuriamo?)
  - *Spazio*: quantità di memoria utilizzata
  - *Banda*: quantità di bit spediti (algoritmi distribuiti)

Come misurare l'efficienza temporale

- Vi sono due strategie principali da prendere in considerazione
- **Misura dipendente dall'architettura hardware/software**
  - Cattura le proprietà delle architetture (particolari istruzioni o supporti)
  - Tipicamente utili in caso di ottimizzazioni di basso livello
  - Si utilizza il concetto di *wall-clock time*: il tempo effettivamente impiegato per eseguire un algoritmo su un dato input
- **Misura indipendente**
  - Si concentra di più sull'algoritmo in sé
  - Più generale, può catturare le caratteristiche dell'algoritmo su più input

Confronto di tempi d'esecuzione

- Per ciascuna funzione $f(n)$ e intervallo di tempo t, si determini la dimensione n più grande che può essere risolta in tempo t, assumendo che l'algoritmo per risolvere il problema richieda $f(n)$ microsecondi.

	1 secondo	1 minuto	1 ora	1 giorno	1 mese	1 anno	1 secolo
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Confronto di tempi d'esecuzione

- Per ciascuna funzione $f(n)$ e intervallo di tempo t, si determini la dimensione n più grande che può essere risolta in tempo t, assumendo che l'algoritmo per risolvere il problema richieda $f(n)$ microsecondi.

	1 secondo	1 minuto	1 ora	1 giorno	1 mese	1 anno	1 secolo
$\lg n$	$2^{10\text{^}6}$	$2^{6\cdot10\text{^}6}$	$2^{36\cdot10\text{^}8}$	$2^{864\cdot10\text{^}8}$	$2^{25920\cdot10\text{^}8}$	$2^{315360\cdot10\text{^}8}$	$2^{31556736\cdot10\text{^}8}$
$\sqrt{n}$	10^{12}	$36\cdot10^{14}$	$1296\cdot10^{16}$	$746496\cdot10^{16}$	$6718464\cdot10^{18}$	$994519296\cdot10^{18}$	$995827586973696\cdot10^{16}$
n	10^{6}	$6\cdot10^{7}$	$36\cdot10^{8}$	$864\cdot10^{8}$	$2592\cdot10^{9}$	$31536\cdot10^{9}$	$31556736\cdot10^{8}$
$n \lg n$	62746	2801417	133378058	2755147513	71870856404	$797634\cdot10^{6}$	$686547\cdot10^{8}$
n^2	1000	7745	60000	293938	1609968	5615692	56175382
n^3	100	391	1532	4420	13736	31593	146677
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Confronto di tempi d'esecuzione