

# Algoritmi e Strutture Dati

## Introduzione

Alberto Montresor and Davide Rossi

Università di Bologna

23 settembre 2024

This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License.



# Sommario

- 1 Introduzione
- 2 Problemi e algoritmi
  - Primi esempi
  - Pseudo-codice
- 3 Valutazione
  - Efficienza
  - Correttezza
- 4 Conclusioni

# Introduzione

## Problema computazionale

Dati un dominio di input e un dominio di output, un *problema computazionale* è rappresentato dalla *relazione matematica* che associa ogni elemento del dominio di input ad uno o più elementi del dominio di output.

## Algoritmo

Dato un problema computazionale, un *algoritmo* è un procedimento effettivo, espresso tramite un insieme di *passi elementari* ben specificati in un sistema *formale* di calcolo, che risolve il problema in tempo *finito*.

## Algoritmo

Un algoritmo è una procedura di calcolo ben definita che prende un certo valore, o un insieme di valori come input e genera, in un tempo finito, un valore, o un insieme di valori, come output. Un algoritmo è quindi una sequenza di passi computazionali che trasforma l'input nell'output.

Algoritmo → strumento per risolvere un problema computazionale ben definito

# Un po' di storia

- Papiro di Rhind o di Ahmes (1850BC): algoritmo del contadino per la moltiplicazione
- Algoritmi di tipo numerico furono studiati da matematici babilonesi ed indiani
- Algoritmi in uso fino a tempi recenti furono studiati dai matematici greci più di 2000 anni fa
  - Algoritmo di Euclide per il massimo comune divisore
  - Algoritmi geometrici (calcolo di tangenti, sezioni di angoli, ...)



[https://en.wikipedia.org/wiki/Rhind\\_Mathematical\\_Papyrus](https://en.wikipedia.org/wiki/Rhind_Mathematical_Papyrus)

# Origine del nome

## Abu Abdullah Muhammad bin Musa al-Khwarizmi

- È stato un matematico, astronomo, astrologo e geografo
- Nato in Uzbekistan, ha lavorato a Baghdad
- Dal suo nome: **algoritmo**



## Algoritmi de numero indorum

- Traduzione latina di un testo arabo ormai perso
- Ha introdotto i numeri indiani (arabi) nel mondo occidentale
- Dal numero arabico **sifr** = 0: zephirum → zevero → zero, ma anche cifra



[https://en.wikipedia.org/wiki/Muhammad\\_ibn\\_Musa\\_al-Khwarizmi](https://en.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Khwarizmi)

# Origine del nome

## Abu Abdullah Muhammad bin Musa al-Khwarizmi

- È stato un matematico, astronomo, astrologo e geografo
- Nato in Uzbekistan, ha lavorato a Baghdad
- Dal suo nome: **algoritmo**



## Al-Kitab al-muhtasar fi hisab al-gabr wa-l-muqabala

- La sua opera più famosa (820 d.C.)
- Tradotta in latino con il titolo:  
*Liber algebrae et almucabala*
- Dal suo titolo: **algebra**



[https://en.wikipedia.org/wiki/Muhammad\\_ibn\\_Musa\\_al-Khwarizmi](https://en.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Khwarizmi)

# Sommario

1 Introduzione

2 Problemi e algoritmi

- Primi esempi
- Pseudo-codice

3 Valutazione

- Efficienza
- Correttezza

4 Conclusioni

# Problemi computazionali: esempi

## Esempio: Minimo

Il minimo di un insieme  $S$  è l'elemento di  $S$  che è minore o uguale ad ogni elemento di  $S$ .

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

## Esempio: Ricerca

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

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

# Algoritmi: esempi

## Algoritmo: 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.

## Algoritmo: 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.

# Problemi

Le descrizioni precedenti presentano diversi problemi:

- **Descrizione**

- Descritti in linguaggio naturale, imprecisi
- Abbiamo bisogno di un linguaggio più formale

- **Valutazione**

- Esistono algoritmi “migliori” di quelli proposti?
- Dobbiamo definire il concetto di migliore

# Come descrivere un algoritmo

- È necessario utilizzare una descrizione il più possibile formale
- Indipendente dal linguaggio: “**Pseudo-codice**”
- Particolare attenzione va dedicata al livello di dettaglio
  - Da una ricetta di canederli, leggo:  
“... amalgamate il tutto e fate riposare un quarto d'ora...”
  - Cosa significa “amalgamare”? Cosa significa “far riposare”?
  - E perché non c’è scritto più semplicemente “prepara i canederli”?

## Esempio: pseudo-codice

---

```
int min(int[] S, int n)
```

---

for  $i = 1$  to  $n$  do

    boolean  $isMin = \text{true}$

    for  $j = 1$  to  $n$  do

        if  $i \neq j$  and  $S[j] < S[i]$

            then

$isMin = \text{false}$

    if  $isMin$  then

        return  $S[i]$

---



---

```
int lookup(int[] S, int n, int v)
```

---

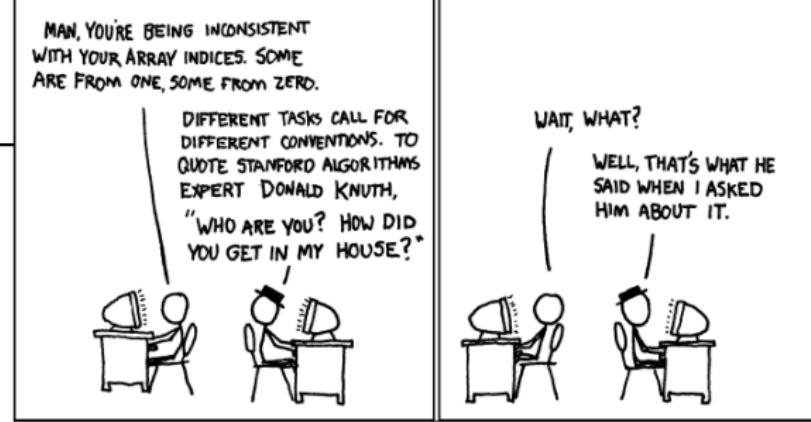
for  $i = 1$  to  $n$  do

    if  $S[i] == v$  then

        return  $i$

return 0

---



# Pseudo-codice

- $a = b$
- $a \leftrightarrow b \equiv$   
 $\quad tmp = a; a = b; b = tmp$
- $T[] A = \text{new } T[1 \dots n]$
- $T[][] B = \text{new } T[1 \dots n][1 \dots m]$
- int, float, boolean, int
- and, or, not
- ==, ≠, ≤, ≥
- +, −, ·, /,  $\lfloor x \rfloor$ ,  $\lceil x \rceil$ , log,  $x^2$ , ...
- iif(*condizione*,  $v_1$ ,  $v_2$ )
- if *condizione* then istruzione
- if *condizione* then istruzione1  
else istruzione2
- while *condizione* do istruzione
- foreach *elemento* ∈ *insieme* do  
istruzione
- return
- % commento

# Pseudo-codice

- **for** *indice* = *estremoInf* **to** *estremoSup* **do** *istruzione*

```
int indice = estremoInf
while indice ≤ estremoSup do
    istruzione
    indice = indice + 1
```

- **for** *indice* = *estremoSup* **downto** *estremoInf* **do** *istruzione*

```
int indice = estremoSup
while indice ≥ estremoInf do
    istruzione
    indice = indice - 1
```

- RETTANGOLO *r* = **new** RETTANGOLO
- *r.altezza* = 10
- **delete** *r*
- *r* = **nil**

---

RETTOANGOLO

---

int lunghezza  
int altezza

---

## Convenzioni dello pseudocodice

L'istruzione return restituisce immediatamente il controllo al punto in cui la procedura chiamante ha eggettato la chiamata

A → array di valori da ordinare

n → numero di valori da ordinare  
(numero elementi di A)

- Il primo elemento dell'array ha indice 1 (No 0)

# Sommario

1 Introduzione

2 Problemi e algoritmi

- Primi esempi
- Pseudo-codice

3 Valutazione

- Efficienza
- Correttezza

4 Conclusioni

# Come valutare l'algoritmo

## Risolve il problema in modo **efficiente**?

- Dobbiamo stabilire come valutare se un programma è efficiente
- Alcuni problemi non possono essere risolti in modo efficiente
- Esistono soluzioni “ottime”: non è possibile essere più efficienti

## Risolve il problema in modo **corretto**?

- Dimostrazione matematica, descrizione “informale”
- Nota: Alcuni problemi non possono essere risolti
- Nota: Alcuni problemi vengono risolti in modo approssimato

## Algoritmo corretto

Un algoritmo si dice corretto relativamente a un problema computazionale se,  $\forall$  istanza del problema fornita in input, esso termina, cioè conclude la sua computazione, e produce, in output la soluzione corretta per la data istanza del problema.

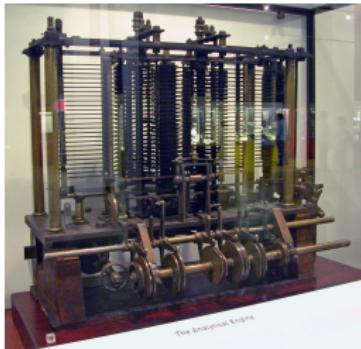
Un algoritmo corretto risolve il problema computazionale dato

Un algoritmo non corretto potrebbe non terminare affatto per qualche istanza di input o potrebbe terminare fornendo un risultato errato.

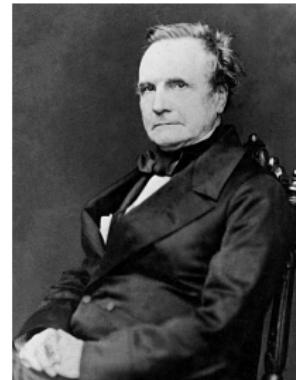
# Charles Babbage

## Passages from the Life of a Philosopher, Charles Babbage, 1864

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 calculation can these results be arrived at by the machine in the shortest time?**



Modello della macchina analitica, Museo di Londra, foto Bruno Barral



Charles Babbage, 1860

# Valutazione algoritmi – Efficienza

## Analisi di costo di un algoritmo

Analisi delle **risorse** impiegate da un algoritmo per risolvere un problema, in funzione della **dimensione** e dalla **tipologia** dell'input

### Risorse

- **Tempo**: tempo impiegato per completare l'algoritmo
  - Misurato con il cronometro?
  - Misurato contando il numero di operazioni rilevanti?
  - Misurato contando il numero di operazioni elementari?
- **Spazio**: quantità di memoria utilizzata
- **Banda**: quantità di bit spediti (algoritmi distribuiti)

Analizzare un algoritmo → prevedere che risorse richiederà

Si possono prendere in considerazione risorse come la memoria, la larghezza di banda nelle comunicazioni o il consumo di energia, ma più frequentemente si è interessati a misurare il tempo di elaborazione

Modello della tecnologia di implementazione che sarà utilizzata

Generico Modello di calcolo a un processore → Modello Random Access Machine

- Le istruzioni sono eseguite una dopo l'altra, non ci sono operazioni contemporanee
- Ogni istruzione richiede la stessa qtà di tempo di qualsiasi altra istruzione
- Ogni accesso ai dati, ovvero l'utilizzo del valore di una variabile o la sua memorizzazione in una variabile, richiede la stessa qtà di tempo di qualsiasi altro accesso ai dati

# Definizione di tempo

**Tempo  $\equiv$  wall-clock time**

Il tempo effettivamente impiegato per eseguire un algoritmo

Dipende da troppi parametri:

Dimensione dell'input

- bravura del programmatore
- linguaggio di programmazione utilizzato
- codice generato dal compilatore
- processore, memoria (cache, primaria, secondaria)
- sistema operativo, processi attualmente in esecuzione

Dobbiamo considerare una rappresentazione più astratta!

# Definizione di tempo – A grandi linee

## Tempo $\equiv$ n. operazioni rilevanti

Numero di operazioni "rilevanti", ovvero il numero di operazioni che caratterizzano lo scopo dell'algoritmo.

## Esempio

- Nel caso del minimo, numero di confronti <
- Nel caso della ricerca, numero di confronti ==

Proviamo!

## Tempo di esecuzione

- Il numero di istruzioni e accessi ai dati che vengono eseguiti

Il tempo di esecuzione dell'algoritmo è la somma dei tempi di esecuzione per ogni istruzione eseguita; un'istruzione che richiede  $c_k$  passi e viene eseguita  $m$  volte contribuirà con  $c_k m$  al tempo di esecuzione totale

# Valutazione algoritmi – Minimo

Contiamo il numero di confronti per il problema del **minimo**

---

```
int min(int[] S, int n)
```

---

```
for i = 1 to n do           n
    boolean isMin = true     n
    for j = 1 to n do         n2
        if i ≠ j and S[j] < S[i] then   n2
            isMin = false             n2
    if isMin then               n
        return S[i]                n
```

---

TEMPO DI ESECUZIONE  $n^2$

# Valutazione algoritmi – Minimo

Contiamo il numero di confronti per il problema del **minimo**

---

```
int min(int[] S, int n)
for i = 1 to n do
    boolean isMin = true
    for j = 1 to n do
        if i ≠ j and S[j] < S[i] then
            isMin = false
    if isMin then
        return S[i]
```

---

Algoritmo “naïf”:  $n^2 - n$

Si può fare meglio di così?

# Valutazione algoritmi – Un algoritmo migliore

Contiamo il numero di confronti per il problema del **minimo**

---

```
int min(int[] S, int n)
% Partial minimum
int min = S[1]
for i = 2 to n do
    if S[i] < min then
        % Update partial minimum
        min = S[i]
return min
```

---

# Valutazione algoritmi – Un algoritmo migliore

Contiamo il numero di confronti per il problema del **minimo**

---

```
int min(int[] S, int n)
```

---

% Partial minimum

```
int min = S[1]
```

```
for i = 2 to n do
```

```
    if S[i] < min then
```

```
        % Update partial minimum
```

```
        min = S[i]
```

---

```
return min
```

---

Algoritmo “naïf”:  $n^2 - n$

Algoritmo efficiente:  $n - 1$

# Valutazione algoritmi – Ricerca

Contiamo il numero di confronti per il problema della ricerca

---

```
int lookup(int[] S, int n, int v)
```

---

```
for i = 1 to n do
```

```
    if S[i] == v then  
        return i
```

---

```
return 0
```

---

Algoritmo “naïf”: *n*

Si può fare meglio di così?

Voglio trovare l'elemento *v* nell'array *S*

↳ se lo trova cui ritorna l'indice *i* (posizione in cui si trova in *S*)

↳ se non lo trova ritorna 0

## Algoritmo Efficiente

Quando analizziamo Algoritmi e prendiamo in considerazione il tempo di esecuzione di solito consideriamo il tempo di esecuzione nel caso peggiore, perché?

- 1) Il caso peggiore di un algoritmo fornisce un limite superiore al tempo di esecuzione su un qualsiasi input
- 2) Per alcuni algoritmi il caso peggiore si verifica molto spesso
- 3) Il "caso medio" spesso ha un costo simile a quello peggiore

## Tasso di crescita

Consideriamo soltanto il termine principale di una formula, in quanto i termini di ordine inferiore sono poco significativi per grandi valori di  $n$

$\Theta(n^2)$  è più efficiente di  $\Theta(n^3)$  perché  $\Theta(n^2)$  ha un tasso di crescita inferiore a  $\Theta(n^3)$

Ordine  
di crescita  
del tempo di  
esecuzione

# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”

1	5	12	15	20	23	32
---	---	----	----	----	----	----

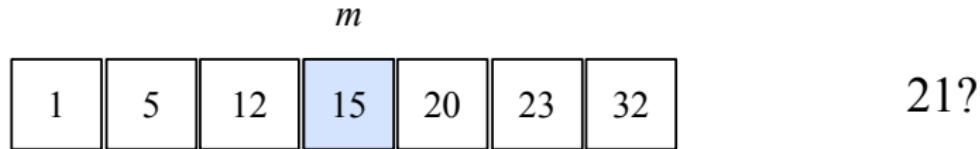
21?

# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”

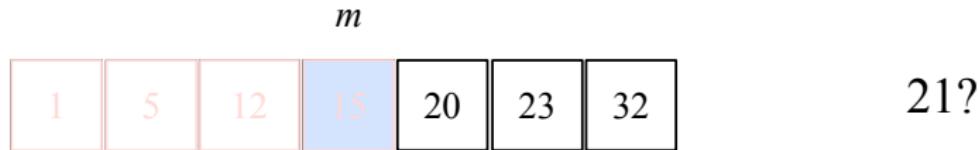


# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”

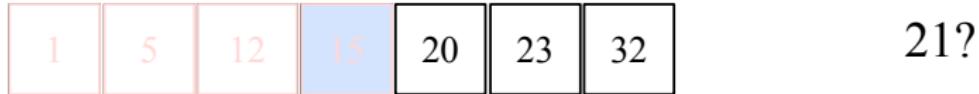


# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”

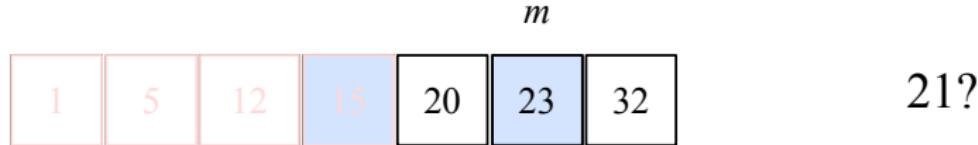


# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”

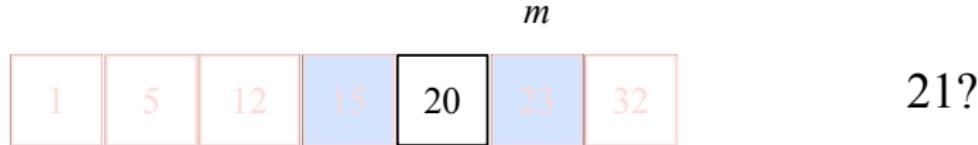


# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”

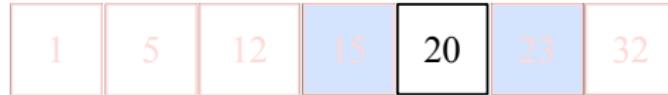


# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”



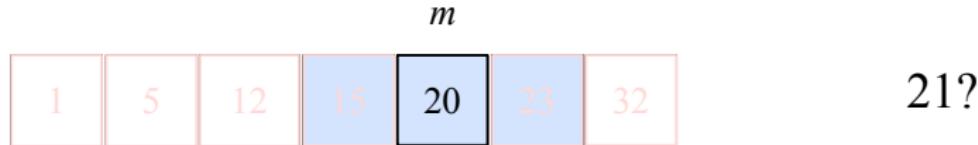
21?

# Valutazione algoritmi – Un algoritmo migliore

## Una soluzione più efficiente

Analizzo l'elemento centrale (indice  $m$ ) del sottovettore considerato:

- Se  $S[m] = v$ , ho trovato il valore cercato
- Se  $v < S[m]$ , cerco nella “metà di sinistra”
- Se  $S[m] < v$ , cerco nella “metà di destra”



# Valutazione algoritmi – Un algoritmo migliore

Contiamo il numero di confronti per il problema della **ricerca**

---

**int binarySearch(int[] S, int v, int i, int j)**

```
if  $i > j$  then          n
|   return 0            n
else                   n
|   int  $m = \lfloor (i + j)/2 \rfloor$     n
|   if  $S[m] == v$  then      n
|   |   return  $m$            n
|   else if  $S[m] < v$  then  n
|   |   return binarySearch( $S, v, m + 1, j$ )  n
|   else                  n
|   |   return binarySearch( $S, v, i, m - 1$ )  n
```

---

Algoritmo “naïf”: **n**

$$T(n/z)$$

$$T(n/z)$$

## Teorema Master per risolvere $T(n/2)$

Il teorema Master  $\rightarrow$  metodo per risolvere equazioni di ricorrenza di algoritmi ricorsivi basati sul metodo divide et impera

### Teorema Master Soluzioni

$$T(n) = \begin{cases} aT(n/b) + g(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Ci sono 3 casi stiche  $\rightarrow$  in base all'aspetto di ricorrenza in base al valore di  $g(n)$ ,  $a, b$

1)  $T(n) = \Theta(n^{\log_b a})$  se  $g(n) = O(n^{\log_b a - \epsilon})$  per qualche  $\epsilon$

2)  $T(n) = \Theta(n^{\log_b a} \log n)$  se  $g(n) = \Omega(n^{\log_b a})$  per qualche  $\epsilon$

3)  $T(n) = \Theta(g(n))$  se  $g(n) = \Omega(n^{\log_b a + \epsilon})$  per qualche  $\epsilon$

Esempio  $g(n) \rightarrow$  soddisfa la condizione di regolarità  $a g(n/b) \leq c \cdot g(n)$  ( $c > 1$ )

$$T(n) = T(n/2) + \Theta(1)$$

Scriviamo l'equazione di ricorrenza in formato generico  $= aT(n/b) + g(n)$

$$a=1$$

$$b=2$$

$g(n) = \Theta(1) \rightarrow$  caso 2 teorema Master

$$\downarrow g(n) = \Theta(n^{\log_2 1}) = \Theta(n^0) = \Theta(1)$$

$$\hookrightarrow T(n) = \Theta(n^{\log_2 a} \cdot \log n) = \Theta(\log n)$$

# Valutazione algoritmi – Un algoritmo migliore

Contiamo il numero di confronti per il problema della **ricerca**

---

```
int binarySearch(int[] S, int v, int i, int j)
```

---

```
if  $i > j$  then
```

```
    return 0
```

```
else
```

```
    int  $m = \lfloor (i + j)/2 \rfloor$ 
```

```
    if  $S[m] == v$  then
```

```
        return  $m$ 
```

```
    else if  $S[m] < v$  then
```

```
        return binarySearch( $S, v, m + 1, j$ )
```

```
    else
```

```
        return binarySearch( $S, v, i, m - 1$ )
```

---

Algoritmo “naïf”:  $n$

Algoritmo efficiente:  
 $2\lceil \log n \rceil$

# Un po' di storia

- 1817: Metodo della bisezione per trovare le radici di una funzione (Bolzano)
- 1946: Prima menzione di binary search (John Mauchly, progettista di ENIAC)
- 1960: Prima versione di binary search che lavora con vettori di dimensione arbitraria (!) (Derrick Henry Lehmer)

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky.

---

*Donald Knuth, The Art of Computer Programming*

[https://hsm.stackexchange.com/questions/2200/  
what-is-the-first-historical-reference-to-the-binary-search-algorithm](https://hsm.stackexchange.com/questions/2200/what-is-the-first-historical-reference-to-the-binary-search-algorithm)  
<https://devopedia.org/binary-search>

# Problemi di overflow

---

```
int binarySearch(int[] S, int v, int i, int j)
```

---

```
if  $i > j$  then
```

```
    return 0
```

```
else
```

```
    int  $m = \lfloor (i + j) / 2 \rfloor$ 
```

```
    if  $S[m] == v$  then
```

```
        return  $m$ 
```

```
    else if  $S[m] < v$  then
```

```
        return binarySearch( $S, v, m + 1, j$ )
```

```
    else
```

```
        return binarySearch( $S, v, i, m - 1$ )
```

---

Algoritmo efficiente:  
 $2\lceil \log n \rceil$

# Valutazione algoritmi – Correttezza

## Invariante

Condizione sempre vera in un certo punto del programma

## Invariante di ciclo

- Una condizione sempre vera all'inizio dell'iterazione di un ciclo
- Cosa si intende per "inizio dell'iterazione"?

## Invariante di classe

- Una condizione sempre vera al termine dell'esecuzione di un metodo della classe

### Invariante di ciclo

L'invariante di ciclo è una proprietà logica che rimane vera in ogni iterazione di un ciclo. Descrive una condizione che il ciclo deve mantenere invariata la sua esecuzione.

# Valutazione algoritmi – Correttezza

Il concetto di **invariante di ciclo** ci aiuta a dimostrare la correttezza di un **algoritmo iterativo**.

- **Inizializzazione** (caso base):

La condizione è vera alla **prima iterazione di un ciclo**

- **Conservazione** (passo induttivo):

Se la condizione è vera prima di un'iterazione del ciclo, allora rimane **vera al termine** (quindi prima della successiva iterazione)

- **Conclusione:**

Quando il ciclo termina, l'invariante deve rappresentare la **"correttezza"** dell'algoritmo

# Valutazione algoritmi – Correttezza

## Invariante

All'inizio di ogni iterazione del ciclo **for**, la variabile *min* contiene il minimo parziale degli elementi  $S[1 \dots i - 1]$ .

---

```
int min(int[] S, int n)
```

---

Inizializzazione

```
int min = S[1]
for i = 2 to n do
    if S[i] < min then
        min = S[i]
return min
```

---

Conservazione

Conclusione

# Valutazione algoritmi – Correttezza

La dimostrazione per induzione è utile anche per gli algoritmi ricorsivi

---

```
int binarySearch(int[] S, int v, int i, int j)
```

---

```
if i > j then
```

```
    return 0
```

```
else
```

```
    int m =  $\lfloor (i + j)/2 \rfloor$ 
```

```
    if S[m] == v then
```

```
        return m
```

```
    else if S[m] < v then
```

```
        return binarySearch(S, v, m + 1, j)
```

```
    else
```

```
        return binarySearch(S, v, i, m - 1)
```

Per induzione sulla dimensione  $n$  dell'input

- Caso base:  
 $n = 0$  ( $i > j$ )
- Ipotesi induttiva:  
vero per tutti gli  $n' < n$
- Passo induttivo:  
dimostrare che è vero per  $n$

# Sommario

- 1 Introduzione
- 2 Problemi e algoritmi
  - Primi esempi
  - Pseudo-codice
- 3 Valutazione
  - Efficienza
  - Correttezza
- 4 Conclusioni

# Altre proprietà

Semplicità, modularità, manutenibilità, espandibilità, robustezza, . . .

- Secondari in un corso di algoritmi e strutture dati
- Fondamentali per un corso di ingegneria del software

## Commento

Alcune proprietà hanno un costo aggiuntivo in termini di prestazioni

- Codice modulare → costo gestione chiamate
- Java bytecode → costo interpretazione

*Progettare algoritmi efficienti è un prerequisito per poter pagare questo costo*

# Binary search, in pillole

## BINÄRY SEARCH

