L'importanza di essere ordinati

corso di laurea in **Matematica**

Informatica Generale, Lezione **6(b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Massimo/minimo di vettori ordinati

Cominciamo con un esempio banale, ma interessante.

Scriviamo le funzioni **`maxV`** e **`minV`** sotto le precondizioni che il vettore di ingresso sia **ordinato**. Quale sarà il costo?

Banalmente il **minimo** è il **primo** elemento,
e il **massimo** è l'**ultimo**.

Questi due problemi diventano **costanti**, cioè $\Theta(1)$.

```
def minV(v):
# REQ: Asc(v)
    return v[0]
```

```
def maxV(v):
# REQ: Asc(v)
    return v[len(v)-1]
```

Ricerca in un vettore ordinato

Cercando un valore x in un **segmento** di **vettore ordinato** $v[inf, sup)$, quale informazione possiamo trarre dal confronto tra x e $v[m]$ per un qualche indice $m \in [inf, sup)$?

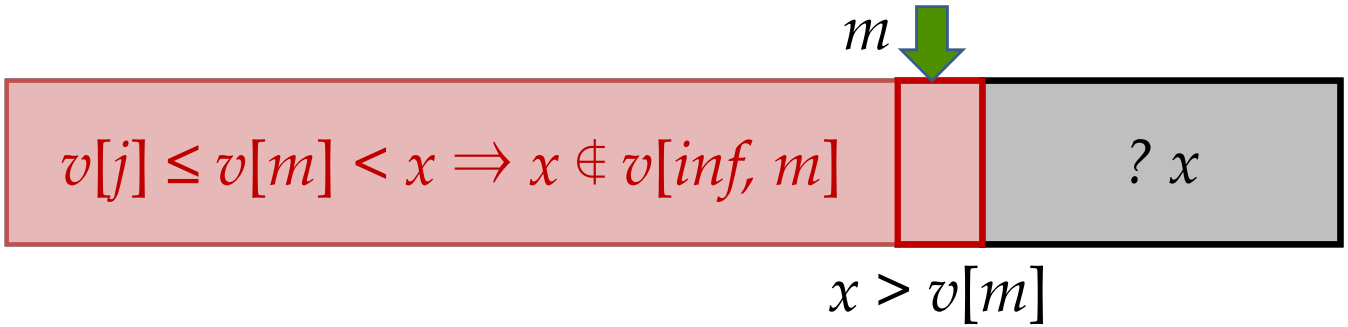
- Se $x = v[m]$ ho **trovato** x in v e ritorno l'indice m.
- Se $x > v[m]$ allora $x > a[j]$ per ogni $j \in [inf, m)$: infatti **Asc**(a) implica che $a[m] \geq a[j]$ e per transitività $x > a[j]$.

Quindi, se $x \in v$, **deve stare** nel **segmento destro** $v[m+1, sup)$.

- Dualmente, se $x < v[m]$ allora $x < a[j]$ per ogni $j \in [m, sup)$.

Quindi, se $x \in v$, **deve stare** nel **segmento sinistro** $v[inf, m)$.

Rimane la questione di **come scegliere** m.



Ricerca binaria: pseudocodice

Osserviamo che l'invariante è banalmente soddisfatto all'ingresso del ciclo perché cerco nell'intervallo [0, len(v)), **cioè tutto il vettore**.

La scelta di *m* è ancora un **segreto di Pulcinella** ma sotto la debole ipotesi m ∈ [*inf*, *sup*) abbiamo:

- $0 \le sup' - inf' = sup - (m+1) \le sup - (inf+1) < sup - inf$, oppure
- $0 \le sup' - inf' = m - inf \le sup - 1 - inf < sup - inf$ $(m < sup)$

quindi *sup – inf* **decresce** e **rimane positiva** se la guardia è vera.

```
def ricercaBinaria(v, x):
# PREC: Asc(v)
# ENS: return j. v[j]==x
    inf, sup = 0, len(v)
    while inf < sup:
      #INV: x ∈ v ⇒ x ∈ v[inf,sup)
       m = dividi(inf, sup)    # inf ≤ m < sup
       if a[m]==x: return m    # x trovato
       if a[m]<x:  inf = m+1   # cerca a destra
            else:  sup = m     # cerca a sinistra
    return -1
```

Ricerca binaria: che fa dividi?

Dovrebbe essere intuitivo che la **scelta migliore** è **dividere in due parti uguali** l'intervallo di ricerca (in assenza di altre "informazioni").

Immaginate di **giocare contro un genio diabolico**: anche se lui vi nasconde l'elemento nella parte più grande, dividendo a metà **minimizzate il rischio** di fare **scelte sbagliate**.

Quindi, è conveniente calcolare il punto medio $m = \left\lfloor \frac{\inf + sup}{2} \right\rfloor$.

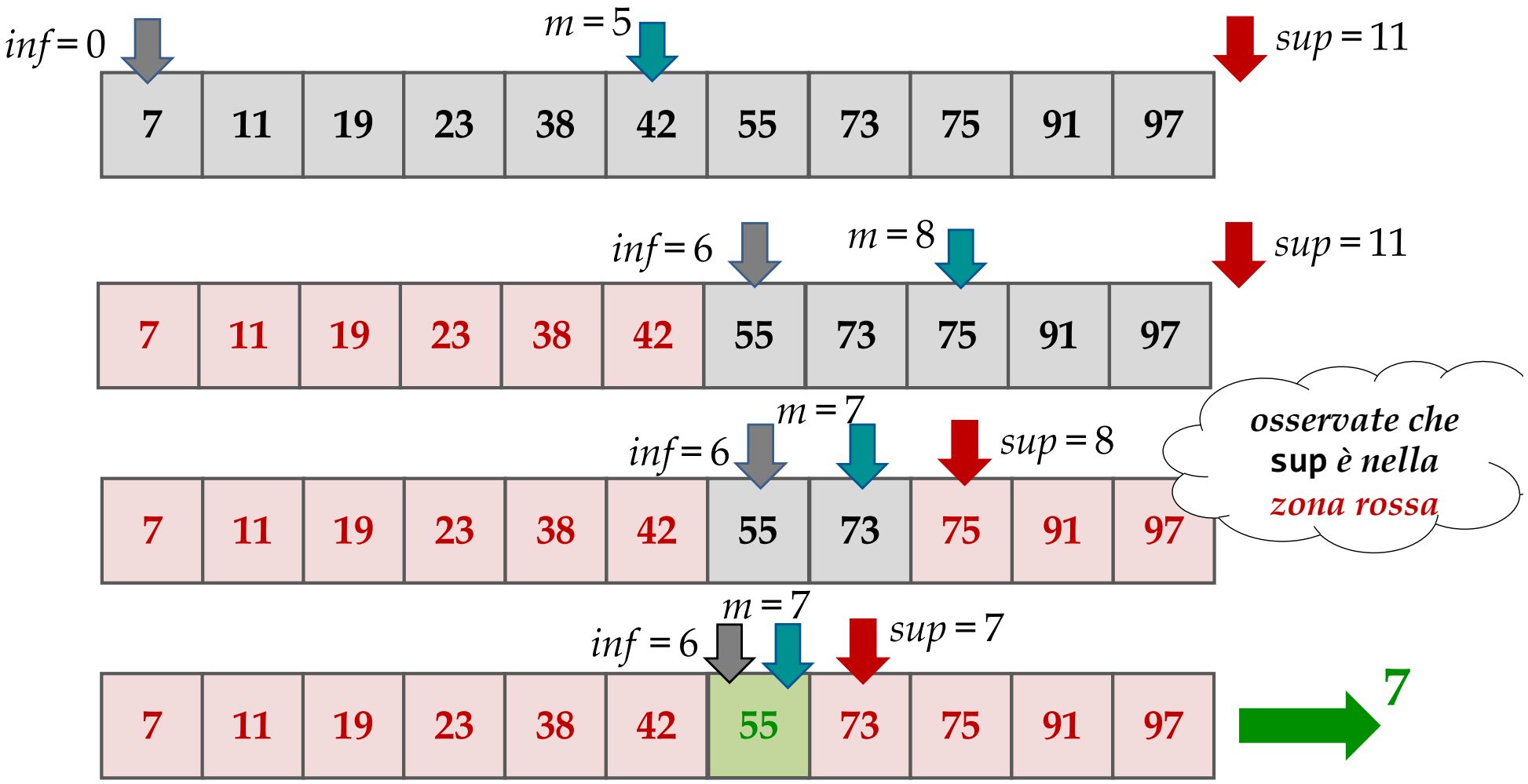
Per la ricerca binaria, viene spesso usata la metafora di una ricerca nel **dizionario** o nell'**elenco telefonico**.

Ma è proprio così che noi cerchiamo in un elenco ordinato?

Usualmente usiamo delle **euristiche**!

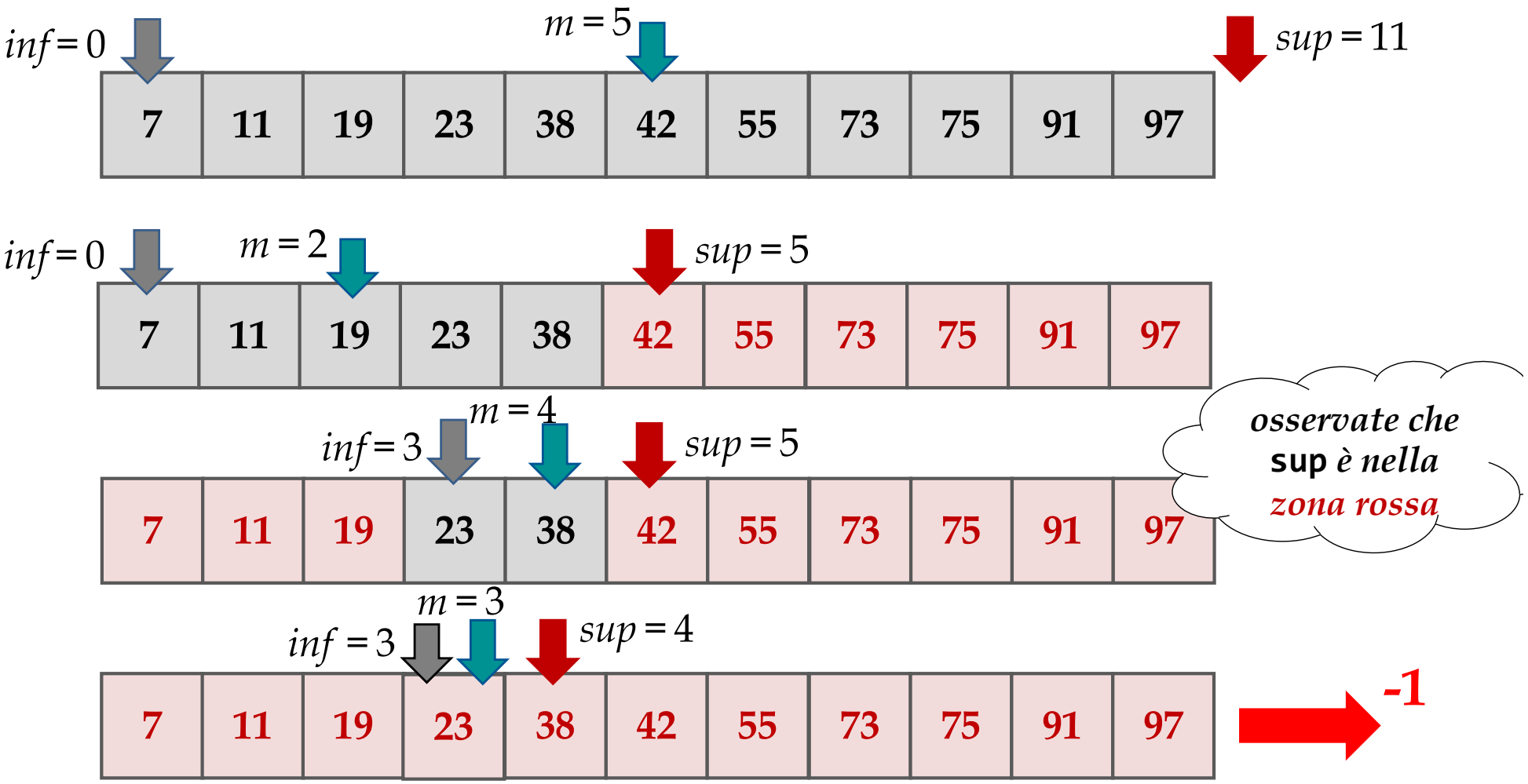
Ricerca binaria: esempio di successo

Cerchiamo il numero… **55**



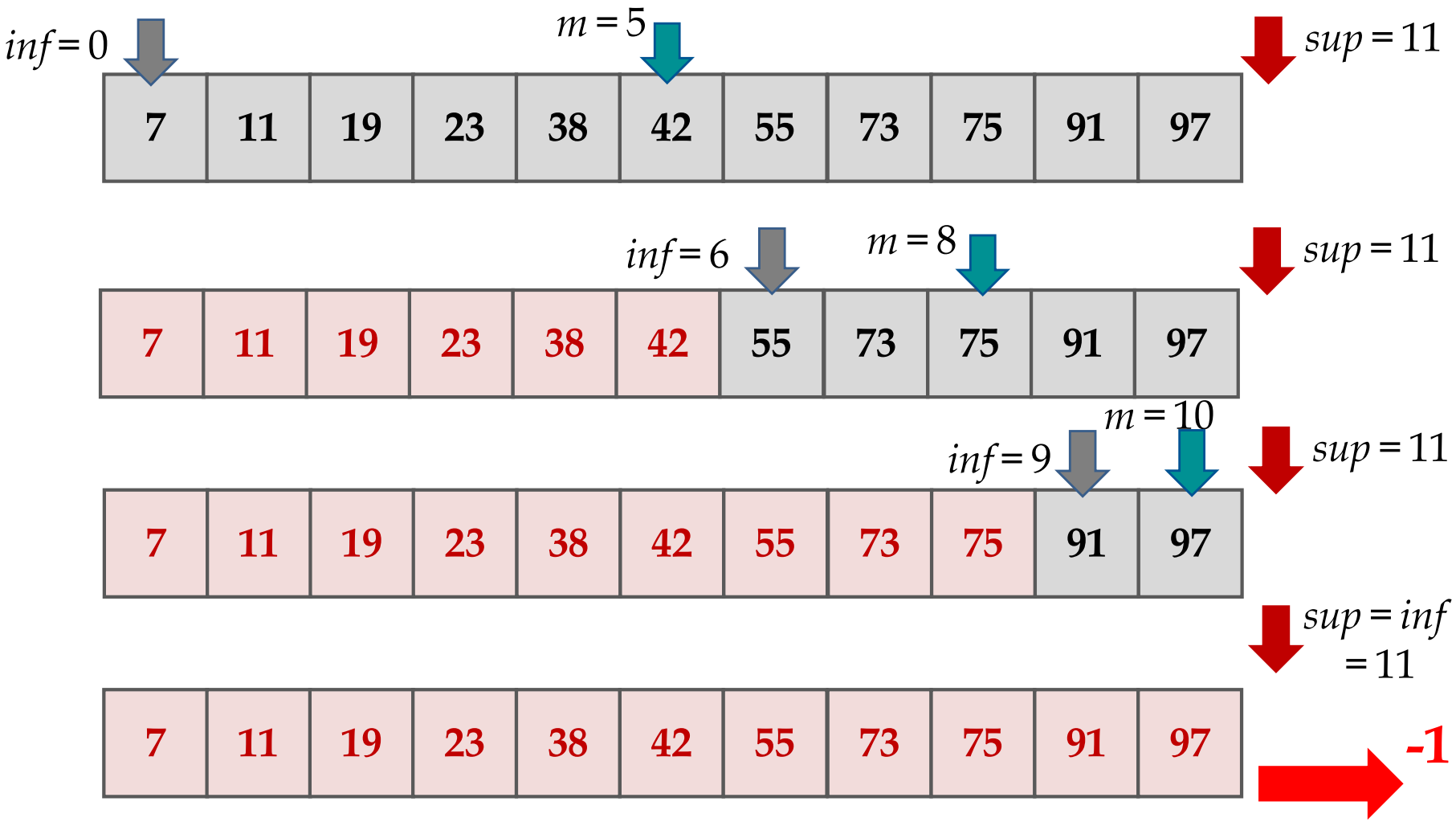
Ricerca binaria: fallimento

Testare varie possibilità! Cerchiamo il numero… **26**



Testing for dummies: casi limite

Testate sempre i casi limite! Cerchiamo il numero… **100**



Ricerca Binaria: caso pessimo

La **funzione di terminazione** $t(inf, sup) = sup - inf$ descrive il numero di elementi nell'intervallo selezionato.

L'intervallo si dimezza ad ogni iterazione, infatti, **caso destro**:

$$sup' - inf' = \sup - \left(\frac{\sup - inf}{2} + 1\right) =$$

$$= \frac{2\sup - \sup - \inf - 2}{2} = \frac{\sup - inf}{2} - 1$$

Attenzione!: ho volontariamente **tralasciato** il fastidio dell'**arrotondamento intero**: in questo caso è **ininfluente** per via del -1, ma a volte occorre fare attenzione per non incartarsi in **intervalli di ampiezza 1**.

Analogamente nel caso sinistro, quando $sup' = m + 1$.

Assumendo abbia all'inizio dimensione $n = 2^k$, la sua dimensione sarà 2^k, 2^{k-1}, 2^{k-2}, …, 2, 1: si **esce** quindi in k passi, cioè $\boldsymbol{\theta(\log_2 n)}$ iterazioni.

Ricerca Binaria: caso medio

Cerchiamo di valutare il caso medio assumendo che il valore da ricercare sia **equiprobabile** e per semplicità $n=2^k$.

Osserviamo che **un solo** elemento (in posizione $n/2$) viene trovato in **1** passo.

Due vengono trovati in **2** passo (posizioni $n/4$, $3n/4$), **quattro** elementi vengono trovati in 2 passi ($n/8$, $3n/8$, $5n/8$, $7n/8$) e così via, fino $\mathbf{2^{k-1}}$ elementi in $\boldsymbol{k}$ passi.

Il valor medio sarà $\frac{1}{n}\sum_{i=1,\ldots,k} i\cdot 2^{i-1}$. Sapendo che $k = \log_2 n$ e:

$$\sum_{i=1,\ldots,k} i\cdot 2^{i-1} = (k-1)\cdot 2^k+1 \quad (*)$$

abbiamo $\frac{1}{n}((\log_2 n - 1)2^{\log n}+1) = \log_2 n - 1 + \frac{1}{n}$

cioè il caso **medio** è come il **pessimo**, a meno di **circa un passo**.

- **Esercizio**: dimostrare per induzione (*)
- **Esercizio**: provare a ricostruire perché vale (*)

Ricerca binaria ricorsiva

La descrizione concettuale della ricerca binaria, si conclude dicendo e "**riapplico il procedimento alla metà sinistra/destra**".

È quindi molto naturale la versione **ricorsiva**, che ha la stessa complessità di quella iterativa.

```
def ricercaBinaria(v, x):
# PREC: Cr(v)
    return ricercaBinariaAux(v, x, 0, len(v))

def ricBinAux(v, x, inf, sup):
    if inf > sup: return -1   # sempre il caso base prima
    m = puntoMedio(inf, sup)  # punto medio
    if a[m]==x: return m      # trovato
    if a[m]<x:
        return ricBinAux(v, x, inf, m) # cerca a sinistra
    return ricBinAux(v, x, m+1, sup)    # cerca a destra
```

usiamo i parametri per "trasmettere" l'intervallo di ricerca

Altre applicazioni: radice quadrata

Il problema di calcolare la **radice quadrata intera** di un numero naturale consiste, dato n, nel trovare un numero r che soddisfi:

$$r^2 \leq n \;\; \& \; n < (r+1)^2.$$

È facile soddisfare $r^2 \leq n$ **sempre** semplicemente **inizializzando** r a 0.

Dopodiché si incrementa r finché è possibile mantenere l'invariante $r^2 \leq n$. Quest'idea ci dà la guardia. Esco con il primo r tale che $r^2 > n$, ma ciò implica che $(r-1)^2 \leq n$ e per la guardia falsificata $r^2 > n$.

Quindi r-1 è il numero che cercavo.

Il semplice programma in figura ha un costo $\Theta(\sqrt{n})$ che è abbastanza **alto** per una **operazione aritmetica**.

Notare che abbiamo fatto **una ricerca lineare** nel **vettore immaginario** dei naturali…

```
def radiceQ(n):
    r = 0
    while r*r ≤ n
        # INV: r² ≤ n
        r = r + 1
    return r - 1
```

Applicazioni della ricerca binaria

Il vettore immaginario dei naturali è però **ordinato** e la **radice è una funzione monotona.**

Idea: confinare il valore $\sqrt{n}$ cercato in un intervallo $[r_1, r_2]$, mantenendo vera la relazione invariante:

$$\varphi \equiv r_1^2 \leq n < r_2^2$$

e **restringere l'intervallo** $[r_1, r_2]$, fino è nella forma $[r_1, r_2 = r_1+1]$, e quindi $r_2 \neq r_1+1$ sarà la nostra guardia.

Restringere l'intervallo significa **diminuire** la grandezza $r_2 - r_1$ che sarà la nostra funzione di terminazione. Per ridurre $r_2 - r_1$ o decremento r_2 oppure incremento r_1 sempre mantenendo vera φ.

Possiamo applicare la **ricerca binaria**, testando $m = \frac{r_1+r_2}{2}$! e cercando nell'intervallo $[r_1, m]$ oppure $[m, r_2]$ a seconda dei risultati. Anche qui non mi devo preoccupare degli intervalli lunghi 1 per come è impostata la guardia.

Rimane il problema delle inizializzazioni, ma $\sqrt{n} \in [0, n]$.

Et voilà le pseudo-code

Il programma segue naturalmente dall'analisi e asserzioni.

Esercizio: ma per calcolare il logaritmo intero? Cosa è preferibile?

Esercizio: e per calcolare 2^n oppure n^n, quali sarebbero i punti deboli di un ragionamento analogo?

```
def radiceQ(n):
    r1, r2 = 0, n
    while r1 + 1 ≠ r2:
        # INV: r1² ≤ n ≤ r2²
        m = puntoMedio(r1, r2)
        if m*m > n then: r2 = m
            else: r1 = m
    return r1
```