

# Efficienza computazionale

## Informatica@SEFA 2018/2019 - Lezione 12

Massimo Lauria <massimo.lauria@uniroma1.it>\*

Venerdì, 26 Ottobre 2018

Usiamo i calcolatori elettronici invece di fare i conti a mano per sfruttare la loro velocità. Tuttavia quando le quantità di dati si fanno molto grandi, anche un calcolatore elettronico comincia ad essere lento. Se il numero di operazioni da fare diventa molto elevato, ci sono due opzioni:

- utilizzare un calcolatore più veloce;
- scrivere un programma più efficiente (i.e. meno operazioni).

Mentre la velocità dei calcolatori migliora con il tempo, questi miglioramenti sono sempre più limitati. Per ridurre il tempo di calcolo non ci sta altro modo che scoprire nuove idee e nuovi algoritmi.

## 1 Trovare uno zero di un polinomio

Considerate il problema di trovare un punto della retta  $\mathbb{R}$  nel quale un dato polinomio  $P$  sia (approssimativamente) 0. Ovvero trovare un  $x \in \mathbb{R}$  per cui  $P(x) = 0$ . Per esempio

$$P(x) = x^5 - 12x^4 + 7x - 6$$

che in Python è calcolato

```
def P(x):  
    return x**5 + 3*x**4 + x**3 - 7*x - 6
```

Ed ha questo aspetto.

---

\*<http://massimolauria.net/courses/infosefa2018/>



assets/plot\_lect10.png

Non è sempre possibile trovare uno zero di una funzione matematica. Tuttavia in certe condizioni è possibile per lo meno trovare uno zero approssimato (i.e. un punto  $x$  per il quale  $|P(x)| \leq \epsilon$  per un  $\epsilon$  piccolo a piacere).

Il polinomio  $P$  ha grado dispari e coefficiente positivo nel termine di grado più alto. Questo vuol dire che

$$\lim_{x \rightarrow -\infty} P(x) \quad \lim_{x \rightarrow \infty} P(x)$$

divergono rispettivamente verso  $-\infty$  e  $+\infty$ .

**Teorema 1** (Teorema degli zeri). *Si consideri una funzione continua  $f : [a, b] \rightarrow \mathbb{R}$ . Se  $f(a)$  è negativo e  $f(b)$  è positivo (o viceversa), allora deve esistere un  $x_0$  con  $a < x_0 < b$  per cui  $f(x_0) = 0$ .*

Verifichiamo subito che ci sono due punti  $a < b$  per cui il polinomio  $P$  cambia di segno.

```
print(P(-2.0),P(1.0))
```

1

```
16.0 -8.0
```

Quindi il teorema precedente vale e ci garantisce che esiste uno zero del polinomio tra  $a$  e  $b$ . Ma come troviamo questo punto? Una possibilità è scorrere tutti i punti tra  $a$  e  $b$  (o almeno un insieme molto fitto di essi).

```
def trova_zero_A(f,a,b):
    Delta= (b - a) / 10000000
    x = a
    steps=1
    while x <= b:
        if -0.00001 < f(x) < 0.00001:
            print("Trovato in {} passi lo zero {}".format(x,steps))
            print(" - f({}) = {}".format(x,f(x)))
            break
        steps +=1
        x += Delta
trova_zero_A(P,-2.0,1.0)
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13

```
Trovato in 3996048 passi lo zero -0.8011859001873525.
- P(-0.8011859001873525) = 7.449223347499867e-06
```

Questo programma trova uno "zero" di  $P$  e ci ha messo 3996048 passi. Possiamo fare di meglio? Cercheremo di trovare un'idea che renda questo calcolo molto più efficiente. Supponiamo che  $P(a) < 0$  e  $P(b) > 0$  (il caso opposto funziona più o meno nella stessa maniera). Possiamo provare a calcolare  $P$  sul punto  $c = (a + b)/2$ , a metà tra  $a$  e  $b$ .

- Se  $P(c) > 0$  allora esiste  $a < x_0 < c$  per cui  $P(x_0) = 0$ ;
- se  $P(c) < 0$  allora esiste  $c < x_0 < b$  per cui  $P(x_0) = 0$ ;
- se  $P(c) = 0$  allora  $x_0 = c$ .

In questo modo dimezziamo ad ogni passo l'intervallo di ricerca.

```
def trova_zero_B(f,a,b):
    passi = 1
    start,end = a,b
    mid = (a + b) / 2
    while not (-0.00001 < f(mid) < 0.00001):
        if f(start)*f(mid) < 0:
            end = mid
        else:
            start = mid
    mid = (start + end)/2
    passi = passi + 1
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13

```

print("Trovato in {} passi lo zero {}".format(mid,passi)) 14
print(" - f({}) = {}".format(mid,f(mid)))                15
trova_zero_B(P,-2.0,1.0)                                  16
                                                         17

```

```

Trovato in 17 passi lo zero -0.8011856079101562.
- P(-0.8011856079101562) = 4.764512533839138e-06

```

Il secondo programma è molto più veloce perché invece di scorrere tutti i punti da  $-2.0$  a  $1.0$ , o comunque una sequenza abbastanza fitta di punti in quell'intervallo, esegue un dimezzamento dello spazio di ricerca. Per utilizzare questa tecnica abbiamo usato il fatto che un polinomio è una funzione continua. Questo è un elemento **essenziale**. Il teorema degli zeri non vale per funzioni discontinue e `trova_zero_B` si basa su di esso.

Se i dati in input hanno della struttura addizionale, questa può essere sfruttata per scrivere programmi più veloci.

Prima di concludere questa parte della lezione voglio sottolineare che i metodi visti sopra possono essere adattati in modo da avere precisione maggiore.

## 2 Ricerca di un elemento in una lista

Come abbiamo visto nella parte precedente della lezione, è estremamente utile conoscere la struttura o le proprietà dei dati su cui si opera. Nel caso in cui in dati abbiano delle caratteristiche particolari, è possibile sfruttarle per utilizzare un algoritmo più efficiente, che però **non è corretto** in mancanza di quelle caratteristiche.

Uno dei casi più eclatanti è la ricerca di un elemento in una sequenza. Per esempio

- cercare un nome nella lista degli studenti iscritti al corso;
- cercare un libro in uno scaffale di una libreria.

Per trovare un elemento  $x$  in una sequenza  $seq$  la cosa più semplice è di scandire  $seq$  e verificare elemento per elemento che questo sia o meno uguale a  $x$ . Di fatto l'espressione `x in seq` fa esattamente questo.

```

def trova(x, seq): 1
    for i,y in enumerate(seq): 2
        if x==y: 3
            print("Trovato l'elemento dopo {} passi.".format(i+1)) 4

```

```
return
print("Non trovato. Eseguiti {} passi.".format(len(seq)))
```

5  
6

Per testare trova ho scritto una funzione

```
test_ricerca(S,sorted=False)
```

che produce una sequenza di 1000000 di numeri compresi tra -2000000 e 2000000, generati a caso utilizzando il generatore **pseudocasuale** di python. Poi cerca il valore 0 utilizzando la funzione S. Se `sorted` è vera allora la sequenza viene ordinata prima che il test incominci.

```
test_ricerca(trova)
```

1

```
Non trovato. Eseguiti 1000000 passi.
```

Per cercare all'interno di una sequenza di  $n$  elementi la funzione di ricerca deve scorrere **tutti** gli elementi. Questo è **inevitabile** in quanto se anche una sola posizione non venisse controllata, si potrebbe costruire un input sul quale la funzione di ricerca non è corretta.

## 2.1 Ordinamento

Nella vita di tutti i giorni le sequenze di informazioni nelle quali andiamo a cercare degli elementi (e.g. un elenco telefonico, lo scaffale di una libreria) sono ordinate e questo ci permette di cercare più velocemente. Pensate ad esempio la ricerca di una pagina in un libro. Le pagine sono numerate e posizionate nell'ordine di enumerazione. È possibile trovare la pagina cercata con poche mosse.

Se una lista `seq` di  $n$  elementi è ordinata, e noi cerchiamo il numero 10, possiamo già escludere metà della lista guardando il valore alla posizione  $n/2$ .

- Se il valore è maggiore di 10, allora 10 non può apparire nelle posizioni successive, e quindi è sufficiente cercarlo in quelle precedenti;
- analogamente se il valore è minore di 10, allora 10 non può apparire nelle posizioni precedenti, e quindi è sufficiente cercarlo in quelle successive.

## 2.2 Ricerca binaria

La ricerca binaria è una tecnica per trovare dati all'interno di una sequenza **seq ordinata**. L'idea è quella di dimezzare ad ogni passo lo spazio di ricerca. All'inizio lo spazio di ricerca è l'intervallo della sequenza che va da 0 a  $\text{len}(\text{seq})-1$  ed  $x$  è l'elemento da cercare.

Ad ogni passo

- si controlla il valore  $v$  a metà dell'intervallo;
- se  $v > x$  allora  $x$  può solo trovarsi prima di  $v$ ;
- se  $v < x$  allora  $x$  può solo trovarsi dopo  $v$ ;
- altrimenti  $v$  è uguale al valore cercato.

```
def ricerca_binaria(x, seq):
    start=0
    end=len(seq)-1
    step = 0
    while start<=end:
        step += 1
        mid = (end + start) // 2
        val = seq[mid]
        if val == x:
            print("Trovato l'elemento dopo {} passi.".format(step))
            return
        elif val < x:
            start = mid + 1
        else:
            end = mid-1
    print("Non trovato. Eseguiti {} passi.".format(step))
```

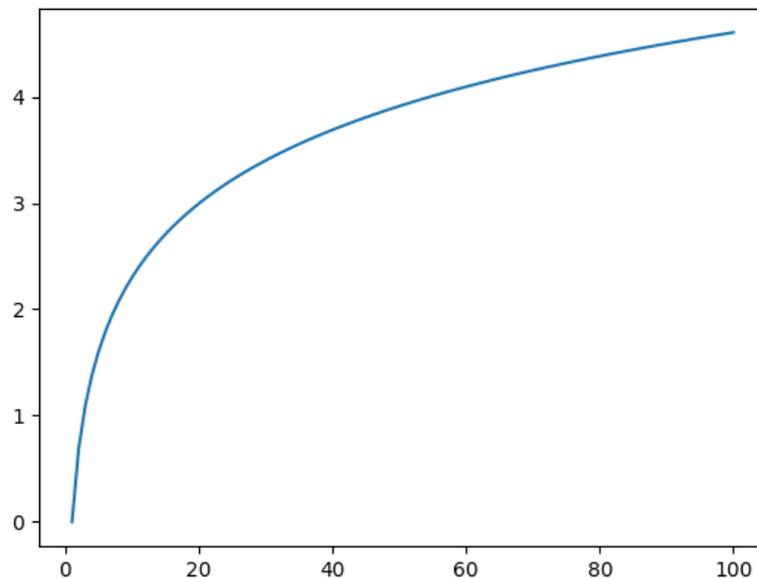
```
test_ricerca(ricerca_binaria, sorted=True)
```

```
Non trovato. Eseguiti 20 passi.
```

La nuova funzione di ricerca è estremamente più veloce. Il numero di passi eseguiti è circa uguale al numero di

- divisioni per due che rendono la lunghezza di  $\text{seq}$  uguale a 1,

ovvero una sequenza di lunghezza  $n$  richiede  $\lceil \log_2 n \rceil$  iterazioni. Mentre la funzione di ricerca che abbiamo visto prima ne richiede  $n$ .



Naturalmente ordinare una serie di valori ha un costo, in termini di tempo. Se la sequenza ha  $n$  elementi e vogliamo fare  $t$  ricerche, ci aspettiamo all'incirca

- Ricerca sequenziale:  $nt$
- Ricerca binaria:  $t \log n + \text{Ord}(n)$

operazioni, dove  $\text{Ord}(n)$  è il numero di passi richiesti per ordinare  $n$  numeri. Quindi se i dati non sono ordinati fin dall'inizio, si può pensare di ordinarli, a seconda del valore di  $t$  e di quanto costi l'ordinamento. In generale vale la pena ordinare i dati se il numero di ricerche è abbastanza consistente, anche se non elevatissimo.

*Non stiamo neppure parlando di quanto costi mantenere i dati ordinati in caso di inserimenti e cancellazioni.*