Corso di laurea in Informatica
Introduzione agli Algoritmi

# Il problema della ricerca

**Angelo Monti**


SAPIENZA
UNIVERSITÀ DI ROMA

In informatica esistono alcuni problemi particolarmente rilevanti, poiché:

- si incontrano in una grande varietà di situazioni reali;

- appaiono come sottoproblemi da risolvere nell'ambito di problemi più complessi.

Uno di questi problemi è la ricerca di un elemento in un insieme di dati (ad es. numeri, cognomi, ecc.).

- Definiamo più formalmente tale problema descrivendone l'input e l'output:

**INPUT**: una lista di $n$ numeri ed un valore $x$.
**OUTPUT**: un indice $i$ tale che $lista[i] = x$ oppure un particolare valore $None$ se il valore $x$ non è presente nella lista.

# La ricerca sequenziale

Un primo semplice algoritmo è basato su questa idea:

- Ispezioniamo uno alla volta gli elementi del vettore
- Confrontiamo ciascun elemento con $x$
- Se troviamo $x$ restituiamo **il suo indice**, terminando l'algoritmo
- Se $x$ non è contenuto nel vettore, alla fine della scansione restituiamo **None**.

## Ricerca lineare

L'algoritmo scorre sequenzialmente la lista degli elementi fino a trovare quello desiderato o fino a determinare che non è presente.

```
def ricerca(lista, x):
    """ Esegue una ricerca lineare per trovare l'indice di un
        elemento in una lista.
    Parametri:   La lista e l'elemento x da ricercare.
    Ritorna: L'indice dell'elemento se trovato, altrimenti None
        """
    n= len(lista)
    for i in range(n):
        if lista[i] == x:
            return i
    return None
```

$$T(n) = O(n)$$

Analizziamo la complessità della ricerca lineare, considerando i due scenari classici: caso ottimo, caso pessimo.

- **caso ottimo.** Si verifica quando l'elemento da cercare è il primo della lista. Solo il primo confronto è necessario per trovare l'elemento. la complessità è $O(1)$, poiché la ricerca termina immediatamente dopo un unico confronto.
- **caso pessimo.** Si verifica quando l'elemento da cercare non è presente nella lista. In questo caso si devono confrontare tutti gli elementi della lista. La complessità è $\Theta(n)$ dove $n$ è il numero totale di elementi della lista.

Nei casi, come questo, in cui non sia possibile determinare un valore stretto per il costo computazionale, ed in cui il caso migliore e quello peggiore si discostano, è naturale domandarsi quale sia il costo computazionale dell'algoritmo nel **caso medio** che rappresenta la situazione in cui l'elemento cercato si trova in una posizione casuale nella lista.

**Ipotesi:** Supponiamo che che l'elemento cercato abbia probabilità uniforme di trovarsi in ogni posizione della lista. In altre parole, la probabilità che $x$ si trovi in ciascuna posizione è la stessa. In questo caso, il numero medio di confronti necessari per trovare $x$ sarà dato dalla media delle posizioni $1, 2, \ldots, n$:

$$\text{numero atteso di confronti} = \sum_{i=1}^{n} i \cdot \frac{1}{n} = .\frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2} = \Theta(n)$$

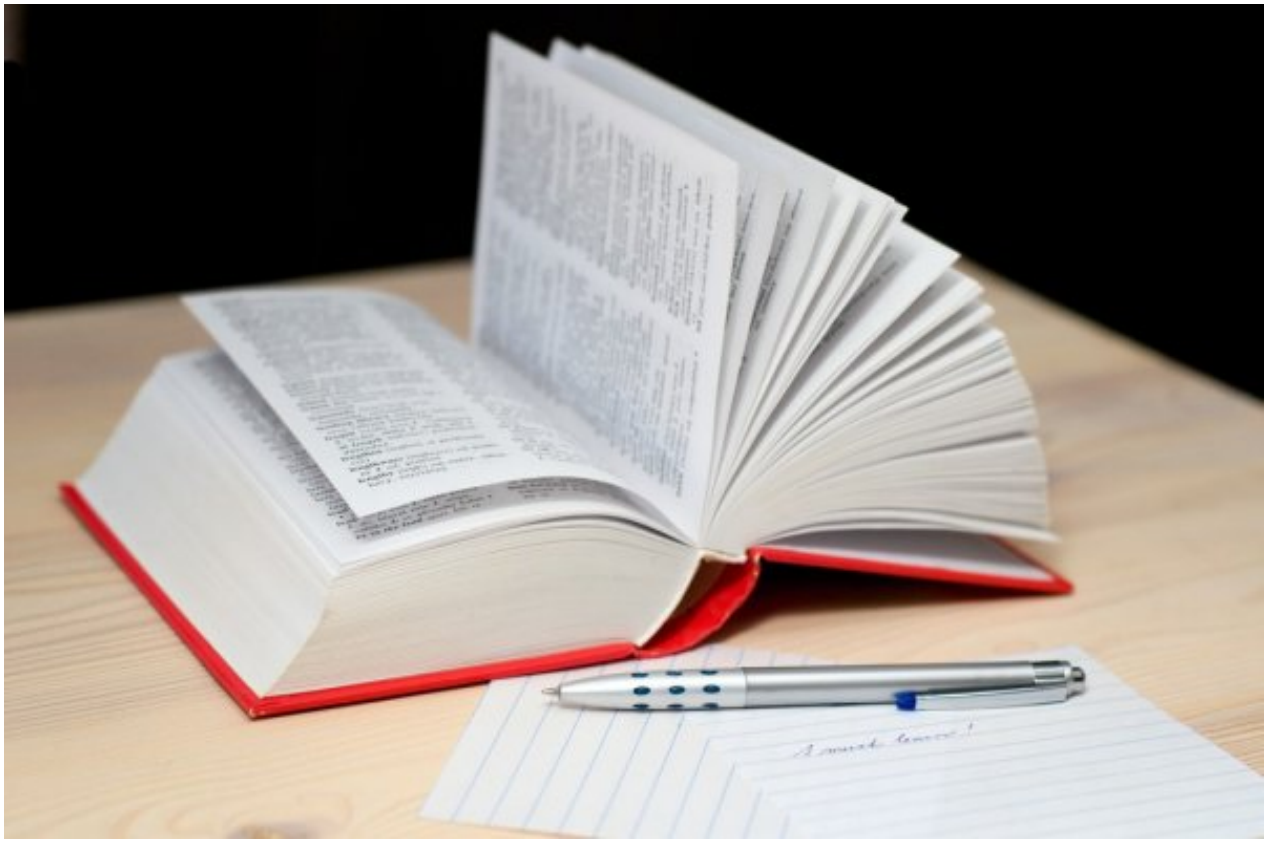
La complessità al caso medio rimane dunque di ordine lineare.

Ricerca in una lista in  python:

```
def ricerca(A, v):
    if v in A:
        return True
    return False
```

L'operatore *in* del Python che verifica se un elemento sia contenuto in un oggetto di tipo *list* con $n$ elementi utilizza una ricerca lineare, perciò ha costo computazionale $O(n)$.
Pertanto: il codice riportato sopra ha costo $O(n)$ nonostante non comprenda (apparentemente!) alcuna istruzione iterativa!

**Ma nella vita di tutti i giorni non ricerchiamo così ….**



## Ricerca Binaria

E' possibile progettare un algoritmo molto più efficiente nel caso in cui la sequenza degli elementi sia ordinata (come sono, ad esempio, i cognomi degli abbonati nell'elenco telefonico).

Come cerchiamo un nome nell'elenco telefonico? Iniziamo sempre e comunque dalla prima lettera dell'alfabeto? Ovviamente no... andiamo direttamente a una pagina dove pensiamo di trovare cognomi che iniziano con la lettera giusta.

Se siamo precisi troviamo il nome nella pagina, altrimenti ci spostiamo in avanti o indietro (di un congruo numero di pagine) a seconda che la lettera del cognome che cerchiamo venga prima o, rispettivamente, dopo quelle delle iniziali dei cognomi contenuti nella pagina.

Ecco di seguito una implementazione in Python del procedimento:

```
def Ricerca_Binaria(lista , x):
    """ Esegue la ricerca binaria in modo iterativo.
    Parametri:la lista ordinata e l'elemento x da cercare.
    Ritorna: L'indice dell'elemento se trovato, altrimenti None
        """
    inizio = 0
    fine = len(lista) - 1
    while inizio <= fine:
        medio = (inizio + fine) // 2
        if lista[medio] == x:
            # Elemento trovato
            return medio
        elif lista[medio] < x:
            # Cerca nella metà di destra
            inizio = medio + 1
        else:
            # Cerca nella metà di sinistra
            fine = medio - 1
    # Elemento non trovato
    return None
```

Calcoliamo ora la complessità di questa implementazione.
La complessità dipende dal numero di iterazioni del ciclo $while$.

- **Caso ottimo** ha complessità $O(1)$ e si ha quando l'elemento di trova nel mezzo dell'insieme ordinato ed è quindi trovato al primo confronto.

- **Caso pessimo** si ha quando l'elemento non è presente nell'insieme ordinato.
  Consideriamo quindi il numero di iterazioni necessarie per ridurre l'intervallo di ricerca a un solo elemento. Ogni iterazione dimezza l'intervallo di ricerca, quindi il numero di elementi da considerare diminuisce come segue:

  - Dopo la prima iterazione, l'intervallo di ricerca contiene $\frac{n}{2}$ elementi.
  - Dopo la seconda iterazione, l'intervallo contiene $\frac{n}{4}$ elementi.
  - Dopo $k$ iterazioni, l'intervallo di ricerca contiene $\frac{n}{2^k}$ elementi.

  Il ciclo continua finché l'intervallo di ricerca non contiene un singolo elemento, cioè quando $\frac{n}{2^k} = 1$. Risolvendo questa equazione per $k$, otteniamo:

  $$\frac{n}{2^k} = 1 \quad \Rightarrow \quad n = 2^k \quad \Rightarrow \quad k = \log_2 n$$

  Pertanto, il numero massimo di iterazioni è $k = \log_2 n$. Concludiamo che la complessità del caso pessimo (quando l'elemento non è presente nella lista) è $\Theta(\log n)$, poiché il numero di iterazioni cresce in modo logaritmico rispetto alla dimensione della lista.

Poiché caso migliore e caso peggiore non hanno lo stesso costo, valutiamo il costo computazionale del **caso medio**, facendo le seguenti assunzioni:

- il numero di elementi è una potenza di 2 (per semplicità dei calcoli, ma è facile vedere che questa assunzione non modifica in alcun modo il risultato finale);
- $x$ è presente nella sequenza (altrimenti si ricade nel caso peggiore);
- tutte le posizioni fra $1$ e $n$ sono equiprobabili.

Domandiamoci ora quante siano le posizioni $n(i)$ raggiungibili alla $i$- esima iterazione:

- con una iterazione si raggiunge la sola posizione $\frac{n}{2}$ cioè $n(1) = 2^0 = 1$
- con due iterazioni si raggiungono le due posizioni $\frac{n}{4}$ e $\frac{3n}{4}$ cioè $n(2) = 2^1 = 2$
- con tre iterazioni si raggiungono le quattro posizioni $\frac{n}{8}$ e $\frac{3n}{8}$ e $\frac{5n}{8}$ e $\frac{7n}{8}$ cioè $n(3) = 2^2 = 4$
- .....

In generale quindi, l'algoritmo di ricerca binaria esegue $i$ confronti se e solo se $x$ si trova in una delle $n(i) = 2^{i-1}$ posizioni raggiungibili con tale numero di confronti.
La probabilità che l'elemento da trovare si trovi in una delle $n(i)$ posizioni è $\frac{n(i)}{n} = \frac{2^{i-1}}{n}$ e il numero medio di confronti è:

$$E = \sum_{i=1}^{\log n} i \cdot \frac{n(i)}{n} = \frac{1}{n} \sum_{i=1}^{\log n} i \cdot 2^{i-1}$$

e sapendo che

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

otteniamo:

$$E = \frac{1}{n}\left((\log_2 n - 1) \cdot 2^{\log_2 n} + 1\right) = \log_2 n - 1 + \frac{1}{n}$$

Cioè, il numero medio di confronti si discosta di meno di un confronto dal numero massimo di confronti.

Con la ricerca binaria ad ogni iterazione **si dimezza il numero degli elementi su cui proseguire l'indagine**.

Questo ci permette di comprendere dove stia la grande efficienza della ricerca binaria: il numero di iterazioni cresce come $\log_2 n$.
Il che significa, ad esempio, che per trovare (o sapere che non c'è) un elemento in una sequenza ordinata di un **miliardo di valori** ($\approx 2^{30}$) bastano circa **30 iterazioni**!

Per completezza riportiamo ora anche la versione ricorsiva dell'algoritmo:

```
def Ricerca_Binaria_R(lista , x, inizio , fine):
    """ Parametri:
          lista:la lista ordinata,
          x: l'elemento  da cercare,
          inizio: l'indice iniziale dell'intervallo di ricerca.
          fine: l'indice finale dell'intervallo di ricerca.
    Ritorna:L'indice dell'elemento se trovato,altrimenti None.
        """
    if inizio > fine:
        # Elemento non trovato
        return None
    medio = (inizio + fine) // 2
    if lista[medio] == x:
        # Elemento trovato
        return medio
    elif lista[medio] < x:
        # Cerca nella metà di destra
        return Ricerca_Binaria_R(lista ,elemento,medio+1,fine)
    else:
        # Cerca nella metà di sinistra
        return Ricerca_Binaria_R(lista ,elemento,inizio ,medio-1)
```

Vedremo poi che la versione ricorsiva e iterativa sono asintoticamente equivalenti per quanto riguarda la complessità di tempo lo stesso non si può dire della complessità di spazio

Corso di laurea in Informatica
Insegnamento di Introduzione agli Algoritmi

# Esercizi per casa


SAPIENZA
UNIVERSITÀ DI ROMA

1. Sia dato un vettore $A$ di interi ed un intero $x$. Il problema consiste nel determinare il numero di volte che $x$ compare in $A$.

   - Progettare un algoritmo che risolva il problema per qualsiasi vettore $A$.
   - Progettare un algoritmo più efficiente del precedente, assumendo che $A$ sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

2. Sia dato un vettore $A$ di interi maggiori o uguali a $0$. Il problema consiste nel determinare se in $A$ sono presenti elementi duplicati.

   - Progettare un algoritmo che risolva il problema per qualsiasi vettore $A$.
   - Progettare un algoritmo più efficiente del precedente, assumendo che $A$ sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

3. Sia dato un vettore $A$ di numeri interi e due valori $a$ e $b$ con $a \leq b$. Il problema consiste nel determinare quanti elementi di $A$ sono compresi nell'intervallo chiuso $[a, b]$.

   - Progettare un algoritmo che risolva tale problema per qualsiasi vettore $A$.
   - Progettare un algoritmo più efficiente del precedente, assumendo che $A$ sia già ordinato e contenga solo valori distinti.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

4. Sia dato un vettore $A$ di numeri interi e un valore $x$. Il problema consiste nel determinare se esistono due elementi distinti di $A$ la cui somma sia esattamente $x$.

   - Progettare un algoritmo che risolva tale problema per qualsiasi vettore $A$.
   - Progettare un algoritmo più efficiente del precedente, assumendo che $A$ sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

5. Sia dato un vettore $A$ di numeri interi e un valore $x$. Il problema consiste nel determinare il minimo valore maggiore di $x$ presente in $A$, o restituire $None$ nel case in cui tale elemento non esista.

   - Progettare un algoritmo che risolva tale problema per qualsiasi vettore $A$.
   - Progettare un algoritmo più efficiente del precedente, assumendo che $A$ sia già ordinato.

   Per ciascun algoritmo, descrivere a parole l'idea alla base della soluzione, scrivere lo pseudocodice e analizzare il tempo di esecuzione asintotica.

6. Sia dato un vettore $A$ di numeri interi ordinati in modo crescente, che è stato ruotato di un numero sconosciuto di posizioni. Progetta un algoritmo per trovare l'indice del punto di rotazione, ovvero l'indice in cui il vettore passa dal valore massimo al valore minimo. L'algoritmo proposto deve avere complessità logaritmica nel numero di elementi presenti in $A$.

7. Sia dato un numero intero $n$. Progetta un algoritmo che calcoli la radice quadrata intera di $n$ ovvero il massimo intero $r$ tale che $^2 \leq n$. L'algoritmo proposto deve avere complessità $O(\log n)$.