Corso di laurea in Informatica
Introduzione agli Algoritmi

Il problema della ricerca

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Nell'informatica esistono alcuni problemi particolarmente rilevanti, poiché essi:

- 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 v.
OUTPUT: un indice i tale che $lista[i] = v$ oppure un particolare valore $None$ se il valore v 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 v
- se troviamo v restituiamo **il suo indice**, terminando l'algoritmo
- se v non è contenuto nel vettore, alla fine della scansione restituiamo **None**.

La ricerca sequenziale

```
def ricercaSeq(A, v):
    n = len(A)
    for i in range(n):
        if A[i] == v:
            return i
    return None
```

$\Theta(1)$

itera $k \leq n$ **volte**

$\Theta(1)$

$\Theta(1)$

$\Theta(1)$

$$T(n) = \Theta(1) + [\Theta(1) + k \cdot \Theta(1)]$$
$$= \Theta(k)$$
$$= O(n)$$

Questo algoritmo ha un costo di:

- $\Theta(n)$ nel caso peggiore (quando, v non è presente nella lista)
- $\Theta(1)$ nel caso migliore (quando v occorre per primo nella lista)

Non abbiamo trovato una stima del costo che sia valida per tutti i casi.

In queste situazioni diremo che il costo dell'algoritmo (in generale, non nel caso peggiore) è un *O(n)*, per evidenziare il fatto che *ci sono input in cui questo valore viene raggiunto, ma ci sono anche input in cui il costo è minore*.

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**.

Facciamo l'ipotesi che gli elementi del vettore A siano distinti e che v possa apparire con uguale probabilità in qualunque posizione, ossia che:

$$P(v \text{ si trova in i-esima posizione}) = \frac{1}{n}$$

Allora il numero medio di iterazioni del ciclo è dato da:

$$\sum_{i=1}^{n} i \cdot \frac{1}{n} = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

il costo computazionale nel caso medio è dunque $\Theta(n)$

Un'ipotesi alternativa è la seguente: supponiamo che tutte le possibili $n!$ permutazioni della sequenza di n numeri siano **equiprobabili**.

Di queste, ve ne saranno un certo numero nelle quali v appare in **prima** posizione, un certo numero nelle quali v appare in **seconda** posizione, ecc.

Il numero medio di iterazioni del ciclo sarà di conseguenza:

$$\sum_{i=1}^{n} i \cdot \frac{\text{numero di permutazioni in cui v è in posizione i}}{\text{numero totale di permutazioni}}$$

Ora, il numero di permutazioni nelle quali v appare nella *i*-esima posizione è uguale al numero delle permutazioni di $n-1$ elementi, cioè $(n-1)!$ dato che fissiamo solo la posizione di uno degli *n* elementi.

$$\text{numero medio di iterazioni} = \sum_{i=1}^{n} i \cdot \frac{(n-1)!}{n!} = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$$

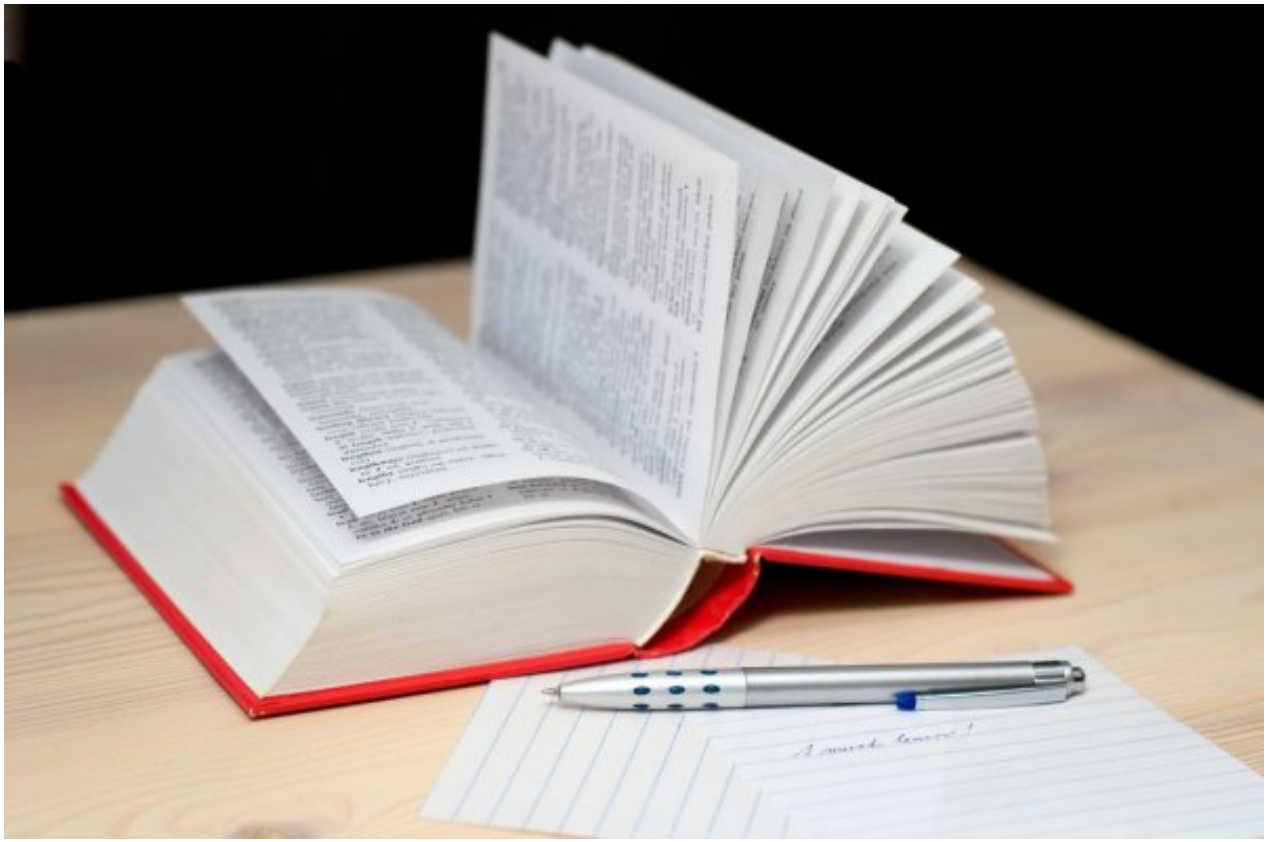
Partendo da due diverse ipotesi di equiprobabilità abbiamo ottenuto lo stesso risultato: il costo computazionale nel caso medio è $\Theta(n)$

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 sequenziale, 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.

RICERCA BINARIA

Un algoritmo che sfrutta questa idea è la ***ricerca binaria***.

Si ispeziona **l'elemento centrale** della sequenza:

- se esso è uguale al valore cercato ci si ferma;
- se il valore cercato è più piccolo si prosegue **nella sola metà inferiore** della sequenza, altrimenti **nella sola metà superiore**.

Una prima considerazione: ad ogni iterazione **si dimezza il numero degli elementi su cui proseguire l'indagine**.

- dopo i iterazioni il numero di elementi su cui ricercare sarà $\frac{n}{2^i}$
- alla peggio ci si ferma quando per i si ha $\frac{n}{2^i} = 1$
- vale a dire $2^i = n \implies i = \log n$
- il numero di iterazioni è dominato da $\log n$

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 $\log n$.

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** bastano circa **30 iterazioni**!

La ricerca binaria

```
def RB(A, v):
    a, b = 0, len(A) - 1
    while a <= b:
        m=(a + b)// 2
        if A[m] == v:
            return m
        elif A[m] > v:
            b = m - 1
        else:
            a = m + 1
    return None
```

Ricerca binaria

Per quanto detto, abbiamo che il ciclo **while** viene eseguito al più $\log n$ volte; pertanto il costo computazionale è:

- $\Theta(\log n)$ nel caso peggiore (l'elemento non c'è);
- $\Theta(1)$ nel caso migliore (l'elemento si trova al primo colpo).

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);
- v **è presente** nella sequenza (altrimenti si ricade nel caso peggiore);
- **tutte le posizioni di** v **fra** 1 **e** n **sono equiprobabili**.

Ricerca binaria

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 $3\frac{n}{4}$, cioè $n(2) = 2^1 = 2$;
- con tre iterazioni si raggiungono le quattro posizioni $\frac{n}{8}$, $3\frac{n}{8}$, $5\frac{n}{8}$, $7\frac{n}{8}$, cioè $n(3) = 2^2 = 4$;

e così via.

In generale quindi, l'algoritmo di ricerca binaria **esegue i iterazioni** se e solo se v si trova in una delle $n(i) = 2^{i-1}$ posizioni raggiungibili con tale numero di iterazioni.

Ricerca binaria

Ricordando che la probabilità che l'elemento da trovare si trovi su una delle $n(i)$ posizioni toccate dalla i-esima iterazione è $\frac{n(i)}{n}$ $\left(\textit{vale a dire } \frac{2^{i-1}}{n}\right)$, il numero medio di iterazioni è:

$$\text{numero medio di iterazioni} = \sum_{i=1}^{\log n} i \cdot \frac{n(i)}{n} = \frac{1}{n} \sum_{i=1}^{\log n} i \cdot 2^{i-1}$$

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

otteniamo: $\frac{1}{n}\left((\log n - 1) \cdot 2^{\log n} + 1\right) = \log n - 1 + \frac{1}{n}$

Ossia, il numero medio di iterazioni si discosta **per meno di un confronto** dal numero massimo di iterazioni!

ESERCIZIO

(dal compito d'esame del 18/2/2021)

Sia dato un vettore A di interi e due valori a e b con $a \leq b$, il problema è quello di sapere quanti elementi di A sono compresi nell'intervallo chiuso $[a, b]$.

•Si progetti un algoritmo per risolvere tale problema su qualsiasi vettore A

•Si progetti un algoritmo più efficiente del precedente assumendo che A sia già ordinato e che contenga solo valori distinti.

Per ciascun algoritmo si descriva a parole l'idea, si scriva lo pseudocodice e si analizzi il tempo di esecuzione asintotica.

ESERCIZIO svolto:
prima parte (vettore disordinato con possibili ripetizioni)

scorro il vettore e conto gli elementi inclusi nell'intervallo $[a, b]$

```
def Conta_in_Intervallo(A, a, b):
    tot = 0
    for i in range(len(A)):
        if a <= A[i] <= b:
            tot += 1
    return tot
```

Costo dell'algoritmo $\Theta(n)$

ESERCIZIO svolto:
seconda parte (vettore ordinato privo di ripetizioni)

Cerco a e b nel vettore con la ricerca binaria.

```
def Conta_in_Intervallo_ordinato(A,a,b):
    Ia = RB(A, a)
    Ib = RB(A, b)
    return Ib - Ia + 1
```

l'algoritmo richiede tempo $\Theta(\log n)$ ma lavora correttamente solo se sia a che b sono entrambi presenti nel vettore A, in caso contrario almeno uno dei due indici è $None$ e si ha errore.

Serve una piccola modifica all'algoritmo di ricerca binaria.

ESERCIZIO svolto:
seconda parte (vettore ordinato privo di ripetizioni)

Consideriamo la versione di ricerca binaria in cui se l'elemento cercato non c'è viene comunque ritornato l'indice m:

```
def RB_modificata (A, v):
    a, b = 0, len(A)-1
    while a <= b:
        m = (a+b)//2
        if A[m] == v:
            return m
        elif A[m] > v:
            b = m - 1
        else:
            a = m + 1
    return m
```

In questo modo per il valore m restituito possono accadere tre cose:

- $A[m] = v$. In questo caso v è in A ed m è la sua posizione
- $A[m] < v$ In questo caso v non è in A ed m è la posizione dell'elemento minore di v più a destra presente in A
- $A[m] > v$ In questo caso v non è in A ed m è la posizione dell'elemento maggiore di v più a sinistra presente in A

ESERCIZIO svolto:
seconda parte (vettore ordinato privo di ripetizioni)

Grazie alla ricerca binaria modificata possiamo risolvere il nostro problema in tempo $\Theta(\log n)$:

```
def Conta_in_Intervallo_ordinato(A, a, b):
    Ia=RB_modificata(A, a)
    if A[Ia] < a: Ia += 1
    Ib=RB_modificata(A, b)
    if A[Ib] > b: Ib -= 1
    return Ib - Ia + 1
```

Corso di laurea in Informatica
Insegnamento di Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO

Sia dato un vettore A di n interi ordinato ed un valore a.
Il problema è quello di sapere quante occorrenze di a sono presenti in A (ovviamente la risposta sarà un intero tra 0 ed n).

• Si progetti un algoritmo per risolvere tale problema in tempo $\Theta(\log n)$.

Per l' algoritmo si descriva a parole l'idea, e poi si scriva lo pseudocodice.