3 Il problema della ricerca

Sommario: *In questo capitolo si affronta un problema basilare, quello della ricerca, e vengono proposte due soluzioni classiche: la ricerca sequenziale (nel caso in cui i dati non siano ordinati) e la ricerca binaria o dicotomica (nel caso in cui i dati siano ordinati).*

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.).
Iniziamo a definire un po' più formalmente tale problema descrivendone l'input e l'output:

- **Input**: un array A di n valori (interi, stringhe, ecc.) ed un valore v;
- **Output**: un indice i tale che $A[i] = v$, oppure -1 se il valore v non è presente nell'array.

3.1 Ricerca sequenziale

Un primo semplice algoritmo che viene in mente consiste nell'ispezionare uno alla volta gli elementi dell'array, confrontarli con v e alla fine restituire il risultato: se l'elemento è presente, ci si interrompe e si restituisce l'indice dell'elemento trovato; altrimenti si restituisce -1 (questo valore è arbitrario, lo abbiamo scelto perché non indica alcun indice).

```
def Ricerca(A, v):
1   i = 0                                   O(1)
2   for i in range (len(A))                 O(1) * al più n volte
3       if (A[i]==v) return i               O(1)
4   return -1                               O(1)
                                            Tot. O(n)
```

Questo algoritmo ha un costo computazionale di $\Theta(n)$ nel caso peggiore (quando, cioè, v non è contenuto nell'array) e di $\Theta(1)$ nel caso migliore (quando v viene incontrato per primo), quindi non abbiamo trovato una stima del costo che sia valida per tutti i casi. In queste situazioni diremo che il costo computazionale 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**. Per fare questo, dobbiamo pensare di calcolare la media dei costi, cioè sommare ciascun costo possibile per la sua probabilità, e poi dividere per il numero dei costi possibili. Non è sempre facile determinare la probabilità di un certo costo e perciò, come in questo caso, facciamo delle ipotesi aggiuntive che ci semplifichino il lavoro.

Facciamo l'ipotesi che *v* possa apparire con uguale probabilità in qualunque posizione, ossia che

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

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

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

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:

$$\text{num. medio di iterazioni} = \sum_{i=1}^{n} i \frac{\text{num. di permutazioni in cui } v \text{ è in posiz. } i}{\text{num. totale di permutazioni}}.$$

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

elementi, cioè $(n-1)!$. Quindi:

$$\textit{num. medio di iterazioni} = \sum_{i=1}^{n} i\frac{(n-1)!}{n!} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}.$$

Facendo due ipotesi di equiprobabilità diverse, abbiamo trovato lo stesso risultato. Dunque, possiamo ragionevolmente pensare che la il costo computazionale del caso medio sia un $\Theta(n)$.

L'operatore *in* del Python che verifica se un elemento è contenuto in un oggetto di tipo *list* ha costo computazionale $O(n)$. Perciò, ad esempio, il seguente frammento di codice:

```
...
if v in A: print('presente')
else: print('assente')
...
```

ha costo $O(n)$ nonostante non comprenda **apparentemente** alcuna istruzione iterativa!

3.2 Ricerca binaria

È possibile progettare un algoritmo più efficiente nel caso in cui la sequenza degli elementi sia ordinata (come sono, ad esempio, le voci in un dizionario).

Procediamo così come facciamo manualmente, andando direttamente a una pagina dove grosso modo pensiamo di trovare le parole che iniziano con la stessa lettera della parola che stiamo

pensando; se siamo precisi troviamo il nome nella pagina, altrimenti ci spostiamo in avanti o indietro (di un congruo numero di pagine) a seconda che l'iniziale della parola che cerchiamo venga prima o, rispettivamente, dopo quelle contenute nella pagina.

Un algoritmo che sfrutta questa idea è la ricerca binaria, mostrata nel seguito. 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.

```
def Ricerca_binaria(A,v):
1   a, b = 0, len(A)-1
2   m = (a + b)//2
3   while (A[m] != v):
4       if (A[m] > v):
5           b = m - 1
6       else:
7           a = m + 1
8       if a > b: return -1
9       m = (a + b)//2
10  return m
```

Una prima considerazione: 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!
Per quanto sopra detto, abbiamo che il ciclo while viene eseguito al più $\Theta(log\ n)$ 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 anche in questo caso il costo computazionale del caso medio, facendo le seguenti assunzioni:

- il numero di elementi è una potenza di 2 (per semplicità di calcolo, 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.

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, 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.

Ricordando che la probabilità che l'elemento da trovare si trovi su una delle $n(i)$ posizioni toccate dalla i-esima iterazione è $n(i)/n$, il numero medio di iterazioni è:

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

Ma ricordando che:

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

otteniamo:

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

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

3.3 Una curiosità: ricerca in tempo costante

Abbiamo detto in questo capitolo che occorre tempo $O(n)$ per eseguire una ricerca di un elemento tra n se essi non sono ordinati, e tempo $O(\log\ n)$ se essi sono invece ordinati.

Tutto ciò è vero sul modello RAM. Ma se cambiassimo modello?

Spostiamoci in un campo completamente diverso. Sia data una lunga sequenza di DNA, e ci chiediamo se un certo frammento sia presente in esso o no. Questo può essere fatto usando la così detta *ibridizzazione*, cioè il processo che unisce due eliche singole di DNA in un'unica elica doppia. È noto infatti che frammenti di singole eliche si attraggono e formano un'elica doppia se essi sono uno il complementare dell'altro (cioè, molto semplicisticamente, se ad ogni nucleotide *A* corrisponde un nucleotide *T* e viceversa, e ad ogni nucleotide *C* corrisponde un nucleotide *G* e viceversa). Per questo scopo, i biologi usano le *sonde*, piccoli frammenti di eliche singole di DNA che hanno una sequenza nota e sono fluorescenti. L'avvenuta ibridizzazione di una sonda in un frammento di DNA sconosciuto evidenzia la presenza della sequenza complementare a quella della sonda nel frammento. Considerando l'ibridizzazione come un singolo passo computazionale su una sorta di ipotetica macchina molecolare, potremmo dire che questo è un algoritmo di ricerca che si può eseguire con costo costante...