Algoritmi di ricerca

Per *ricerca* si intende qui il procedimento di localizzare una particolare informazione in un elenco di dati.

Per esempio:

- cercare il numero di Ermenegildo Rossi nell'elenco del telefono;
- cercare il re di denari in un mazzo di carte mescolate.

Dai precedenti esempi si intuisce che il metodo di ricerca dipende da come le informazioni sono organizzate.

Il problema si può porre in termini generali così:

> *Il problema della ricerca*
> Dato un insieme $L = \{a_1,a_2,...,a_n\}$ di n elementi distinti e un elemento x, determinare se x appartiene all'insieme.

Ricerca lineare

Se non abbiamo alcuna informazione circa l'ordine degli elementi nell'insieme, l'unico metodo per localizzare un particolare elemento è una *ricerca lineare*; cioè si parte dal primo elemento e si procede esaminando uno per uno tutti gli elementi finché non si verifica una delle due condizioni:

- l'elemento cercato è stato individuato nella posizione i ($x = a_i$), *oppure*
- abbiamo scandito tutti gli elementi dell'insieme senza aver trovato l'elemento x.

Algoritmo Ricerca 1:

```
for (i = 0; i < n && a[i] != x; i++)
    ;
if (i < n)
   cout << "Trovato al posto " << i << endl;
else
   cout << "Non trovato" << endl;
```

La seguente è una variante dell'algoritmo precedente, dove *x* viene inserito come (*n*+1)-esimo elemento dell'insieme.

Algoritmo Ricerca 2:

```
a[n] = x;
for (i=0; a[i]!=x; i++);
if (i<n)
   cout << "Trovato al posto " << i << endl;
else
   cout << "Non trovato" << endl;
```

Osserviamo che in quest'ultimo algoritmo dobbiamo eseguire un solo confronto per ciclo invece di due, come nel precedente.

Nel caso più sfavorevole, quando l'elemento *x* non si trova nell'insieme *L*, l'algoritmo Ricerca 1 effettua 2*n* confronti e l'algoritmo Ricerca 2 effettua *n*+1 confronti.

Osserviamo però che le due funzioni 2*n* e *n*+1 sono dello stesso ordine *O(n)*.

Complessità computazionale di un problema

Definizioni:

> Una funzione $f(n)$ ha *ordine* $O(g(n))$ se e solo se esistono due costanti positive c e n' tali che $|f(n)| \leq c|g(n)|$ per ogni $n \geq n'$.
>
> Un algoritmo ha una *complessità* $O(g(n))$ se il tempo di calcolo $t(n)$ sufficiente per eseguire l'algoritmo con ogni *istanza* (l'insieme dei dati su cui è definito il problema) di dimensione *n* ha ordine $O(g(n))$.

Dal punto di vista della complessità i due algoritmi di ricerca esaminati precedentemente sono equivalenti; infatti il tempo di esecuzione cresce sempre linearmente con *n* in entrambi i casi.

La complessità di un algoritmo è funzione della lunghezza dell'istanza.

L'analisi che viene fatta è *asintotica*; cioè si valuta la complessità dell'algoritmo esaminandone il comportamento al limite per valori della lunghezza delle istanze tendenti all'infinito.

Ricerca binaria

Supponiamo che l'insieme $L = \{a_1,a_2,...,a_n\}$ sia *ordinato*, cioè che si abbia $a_i \leq a_{i+1}$ per ogni i compreso tra 1 e n-1. Possiamo in questo caso applicare un algoritmo molto migliore dal punto di vista della complessità computazionale, il cosiddetto algoritmo della *ricerca binaria*.

Algoritmo Ricerca binaria:

```
max = n-1;
min = 0;
trovato = 0;
while (!trovato && max>=min) {
   med = (max+min)/2;
   if (a[med] == x)
      trovato = 1;
   else if (a[med] > x)
      max = med-1;
   else min = med+1;
   }
cout << x;
if (trovato)
   cout << " si trova nell'insieme alla posizione " << med;
else
   cout << " non si trova nell'insieme";
```

Per studiare la complessità dell'algoritmo della ricerca binaria occorre valutare quante volte viene eseguito il ciclo `while` nel caso peggiore, che è quello in cui x non si trova nell'insieme.

Considerando che a ogni iterazione l'insieme è dimezzato, il numero di iterazioni è pari a quante volte un numero n può essere diviso per 2 fino a ridurlo a 0. Questo numero è il più piccolo intero non minore di $1+\log_2 n$; quindi possiamo concludere che l'algoritmo della ricerca binaria ha complessità $O(\log n)$.

Se per esempio dovessimo cercare un elemento in un insieme di 1.000.000 di elementi, nei casi più sfortunati con gli algoritmi Ricerca 1 e Ricerca 2 dovremmo eseguire circa 1.000.000 di iterazioni, mentre con la Ricerca binaria ne dobbiamo effettuare al massimo solamente 21.

È possibile abbassare ulteriormente la complessità? La risposta è negativa; infatti si può dimostrare che, qualunque possa essere l'algoritmo considerato, un numero logaritmico di passi è comunque necessario per poter risolvere il problema della ricerca.

La delimitazione inferiore alla complessità computazionale di un problema

La delimitazione inferiore fornisce la *complessità intrinseca* del problema.

> Una funzione *f(n)* è $\Omega(g(n))$ se e solo se esistono due costanti *c* e *n'* tali che $|f(n)| \geq c|g(n)|$ per ogni $n \geq n'$.
>
> Un problema ha una *delimitazione inferiore* alla complessità $\Omega(g(n))$ se, qualunque sia l'algoritmo di risoluzione, per ogni *n*, esiste una istanza per cui il tempo di calcolo necessario *t(n)* è $\Omega(g(n))$.

$\Omega(g(n))$ rappresenta quindi la complessità intrinseca di un problema che dipende dalle sue caratteristiche strutturali.

Un algoritmo che risolve un problema *P* si dice *ottimale* se valgono le due seguenti condizioni:

- l'algoritmo risolve *P* con un costo di esecuzione *O(g(n))*
- *P* ha una delimitazione inferiore $\Omega(g(n))$

L'algoritmo della ricerca binaria è un esempio di algoritmo ottimale.

Algoritmi di ordinamento

Il problema dell'ordinamento
Dato un insieme L di n elementi, ordinare L.

Bubble sort

```
                                last
                                 ↓
2   1   8   4   7   6   3   5
1   2   8   4   7   6   3   5
1   2   8   4   7   6   3   5
1   2   4   8   7   6   3   5
1   2   4   7   8   6   3   5
1   2   4   7   6   8   3   5
1   2   4   7   6   3   8   5
1   2   4   7   6   3   5   8

                            last
                             ↓
1   2   4   7   6   3   5   8
1   2   4   7   6   3   5   8
1   2   4   7   6   3   5   8
1   2   4   7   6   3   5   8
1   2   4   6   7   3   5   8
1   2   4   6   3   7   5   8
1   2   4   6   3   5   7   8

                        last
                         ↓
1   2   4   6   3   5   7   8
1   2   4   6   3   5   7   8
1   2   4   6   3   5   7   8
1   2   4   6   3   5   7   8
1   2   4   3   6   5   7   8
1   2   4   3   5   6   7   8

                    last
                     ↓
1   2   4   3   5   6   7   8
1   2   4   3   5   6   7   8
1   2   4   3   5   6   7   8
1   2   3   4   5   6   7   8
1   2   3   4   5   6   7   8
```

Numero totale di confronti:

$$(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = n(n\text{-}1)/2 = n^2/2\text{-}n/2$$

$$O(n^2)$$

Insertion sort

```
     newPos
     ↓
2    1    8    4    7    6    3    5                 newValue = 1
2    2    8    4    7    6    3    5
1    2    8    4    7    6    3    5

          newPos
          ↓
1    2    8    4    7    6    3    5                 newValue = 8
1    2    8    4    7    6    3    5

               newPos
               ↓
1    2    8    4    7    6    3    5                 newValue = 4
1    2    8    8    7    6    3    5
1    2    4    8    7    6    3    5

                    newPos
                    ↓
1    2    4    8    7    6    3    5                 newValue = 7
1    2    4    8    8    6    3    5
1    2    4    7    8    6    3    5

                         newPos
                         ↓
1    2    4    7    8    6    3    5                 newValue = 6
1    2    4    7    8    8    3    5
1    2    4    7    7    8    3    5
1    2    4    6    7    8    3    5

                              newPos
                              ↓
1    2    4    6    7    8    3    5                 newValue = 3
1    2    4    6    7    8    8    5
1    2    4    6    7    7    8    5
1    2    4    6    6    7    8    5
1    2    4    4    6    7    8    5
1    2    3    4    6    7    8    5

                                   newPos
                                   ↓
1    2    3    4    6    7    8    5                 newValue = 5
1    2    3    4    6    7    8    8
1    2    3    4    6    7    7    8
1    2    3    4    6    6    7    8
1    2    3    4    5    6    7    8
```

Shell sort

Incr = 4

2 1 8 4 7 6 3 5
2 1 8 4 7 6 3 5
2 1 8 4 7 6 3 5
2 1 3 4 7 6 8 5

Incr = 1

2 1 3 4 7 6 8 5
1 2 3 4 7 6 8 5
1 2 3 4 7 6 8 5
1 2 3 4 7 6 8 5
1 2 3 4 7 6 8 5
1 2 3 4 6 7 8 5
1 2 3 4 6 7 8 5
1 2 3 4 6 7 5 8
1 2 3 4 6 5 7 8
1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

Quicksort

```
                                                    pivot = 4
   start          pivot              finish
   ↓              ↓                  ↓
   2   1   8   4   7   6   3   5
   2   1   8   4   7   6   3   5
   2   1   8   4   7   6   3   5
   2   1   3   4   7   6   8   5
   2   1   3   4   7   6   8   5


start piv finish          start pivot finish
   ↓   ↓   ↓                ↓   ↓        ↓
   2   1   3       4        7   6   8    5
   2   1   3       4        5   6   8    7

   1   2   3       4        5   6   7    8
```

Numero totale di confronti: *O(n* log *n)*
Algoritmo ottimale, in quanto $\Omega(n \log n)$ è una delimitazione inferiore al problema dell'ordinamento.

Classi di complessità

1) Complessità *costante*
O(*1*)
È posseduta dagli algoritmi che eseguono sempre lo stesso numero di operazioni indipendentemente dalla dimensione dei dati.
Es.: inserimento o estrazione dalla testa di una lista concatenata, ecc.

2) Complessità *sottolineare*
O(n^k), con $k<1$

Es.: $O(\sqrt{n})$, $O(\sqrt[3]{n})$, $O(\log n)$

Ricerca binaria, o *logaritmica*, ecc.

3) Complessità *lineare*
O(*n*)
È posseduta dagli algoritmi che eseguono un numero di operazioni sostanzialmente proporzionale ad *n*.
Es.: ricerca sequenziale, somma di numeri di *n* cifre, merge di due file, ecc.

4) Complessità *n log n*
O(*n* log *n*)
Es.: Algoritmi di ordinamento ottimi.

5) Complessità n^k, con $k \geq 2$
O(n^k)
Es.: Bubblesort (O(n^2)), moltiplicazione di due matrici (O(n^3)), ecc.

6) Complessità *esponenziale*
Tutte le classi di complessità elencate precedentemente vengono genericamente considerate come *polinomiali*. Esse sono caratterizzate dal fatto che la dimensione *n* non compare mai come esponente in alcun modo. Quando ciò avviene si parla invece di *complessità esponenziale*.
Es.: un algoritmo che debba produrre tutte le possibili stringhe lunghe *n* di *k* simboli avrà complessità $O(k^n)$, cioè esponenziale.
L'algoritmo della Torre di Hanoi ha complessità $O(2^n)$ (sono necessarie 2^n - 1 mosse).

È necessario guardare con particolare diffidenza agli algoritmi dotati di complessità esponenziale, perché possono richiedere dei tempi di esecuzione proibitivi (se non addirittura astronomici) anche per valori relativamente piccoli di *n* ed indipendentemente dalla velocità dell'elaboratore.

Purtroppo esistono molti problemi, anche di interesse pratico, per i quali non si conoscono ancora algoritmi non esponenziali (problemi *intrattabili*). Es.: problema del commesso viaggiatore.

Una funzione *f(n)* è *limitata superiormente* da una funzione *g(n)* se esiste un intero n_0 tale che, per ogni $n \geq n_0$, $f(n) \leq g(n)$.

Una funzione *polinomiale f(n)* è una funzione limitata superiormente da n^k per un opportuno fissato intero *k*.

Un algoritmo si dice *efficiente* se il suo tempo di esecuzione è limitato superiormente da una funzione polinomiale.

Un problema è *computazionalmente trattabile* se esiste un algoritmo efficiente che lo risolve; altrimenti il problema è *computazionalmente intrattabile*.