Corso di Introduzione agli algoritmi
Prof.ssa Tiziana Calamoneri

Il problema della ricerca

Il problema della ricerca

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:

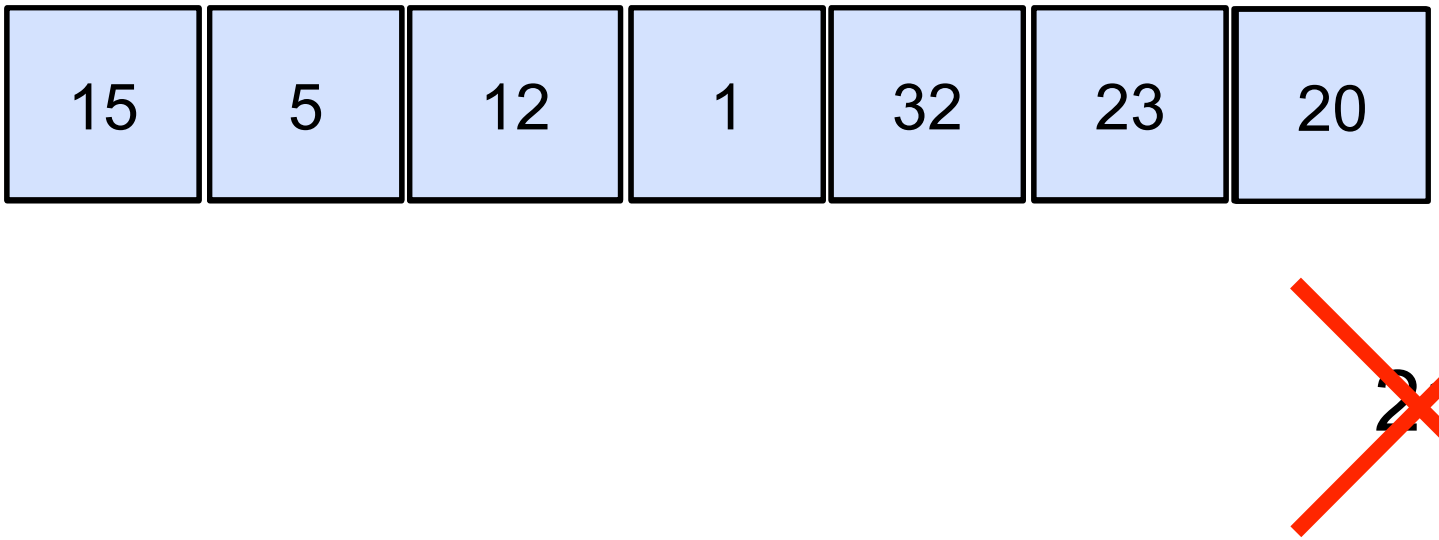
Input: un vettore *A* di *n* numeri ed un valore *v*;

Output: un indice *i* tale che *A[i]* = *v*, oppure un particolare valore ***null*** se il valore *v* non è presente nel vettore.

La ricerca sequenziale (1)

Un primo algoritmo:

- ispezioniamo uno alla volta gli elementi del vettore
- confrontiamo ciascun elemento con *v*
- restituiamo il risultato, interrompendoci appena (non) trovato *v*



La ricerca sequenziale (2)


```
Funzione RicercaSeq (A: vettore; v: intero)
   i ← 1                                   Θ(1)
   while ((i ≤ n) and (A[i] ≠ v))          Θ(1)+k ≤ n volte
         i ← i + 1                         Θ(1)
   if (i ≤ n)                              Θ(1)
         return i                          Θ(1)
   else
         return null                       Θ(1)
```

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

La ricerca sequenziale (3)

Questo algoritmo ha un costo computazionale di:

- $\Theta(n)$ nel caso peggiore (quando, cioè, *v* non è contenuto nel vettore) e
- di $\Theta(1)$ nel caso migliore (quando *v* viene incontrato per primo)
- 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*.

La ricerca binaria (1)

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

- ispezioniamo l'elemento centrale della sequenza
- se esso è uguale al valore cercato ci fermiamo
- se il valore cercato è più piccolo proseguiamo nella sola metà inferiore della sequenza
- altrimenti nella sola metà superiore.

La ricerca binaria (2)


```
Funzione Ricerca_binaria (A: vettore; v: intero)
    a ← 1
    b ← |A|
    m ← (a+b)/2
    while (A[m] ≠ v)
            if (A[m] > v)
                    b ← m – 1
            else
                    a ← m + 1
            if (a > b) return null
            m ← (a+b)/2
    return m
```

La ricerca binaria (3)

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!

In generale, 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).

La ricerca binaria (4)

consideriamo ora una diversa formulazione dell'algoritmo di ricerca binaria (nella quale sono volutamente tralasciati i dettagli per catturarne l'essenza):

```
Ricerca_binaria (A, v)
    se il vettore è vuoto restituisci null
    ispeziona l'elemento A[centrale]
    se esso è uguale a v restituisci il suo indice
    se v < A[centrale]
            esegui Ricerca_binaria (metà sinistra di A, v)
    se v > A[centrale]
            esegui Ricerca_binaria (metà destra di A, v)
```

La ricerca binaria (5)

L'aspetto cruciale di questa formulazione risiede nel fatto che l'algoritmo risolve il problema ***"riapplicando" se stesso su un sottoproblema*** (una delle due metà del vettore).

Questa tecnica si chiama ***ricorsione***.

La ricorsione

Funzioni matematiche ricorsive

- Una funzione matematica è detta ricorsiva quando la sua definizione è espressa in termini di se stessa.
- Esempio:

  $n! = n*(n - 1)!$ Se $n > 0$

  $0! = 1$

  - la prima riga determina il meccanismo di calcolo ricorsivo vero e proprio, ossia stabilisce come, conoscendo il valore della funzione per un certo numero intero, si calcola il valore della funzione per l'intero successivo.
  - La seconda riga della definizione invece determina un altro aspetto fondamentale, il ***caso base***: in questo caso esso indica che, per il numero 0, il valore della funzione fattoriale è 1.
- N.B. una funzione mat. deve sempre avere un caso base!

Algoritmi ricorsivi (1)

Nel campo degli algoritmi vi è un concetto del tutto analogo, quello degli algoritmi ricorsivi:

- un algoritmo è detto ricorsivo quando è espresso in termini di se stesso.

Un algoritmo ricorsivo ha sempre queste proprietà:

- la soluzione del problema complessivo è costruita risolvendo (ricorsivamente) uno o più sottoproblemi di dimensione minore e successivamente producendo la soluzione complessiva mediante combinazione delle soluzioni dei sottoproblemi;
- la successione dei sottoproblemi, che sono sempre più piccoli, deve sempre convergere ad un sottoproblema che costituisca un caso base (detto anche ***condizione di terminazione***), incontrato il quale la ricorsione termina.

Algoritmi ricorsivi (2) - fattoriale

Esempio: Calcolo del fattoriale di un intero dato *n.*

```
Funzione Fattoriale_Iter (n: intero non negativo)
    if (n > 0)
        fatt ← 1
        for i=1 to n do
            fatt ← fatt*i
        return fatt
    else return 1
```

```
Funzione Fattoriale_Ric (n: intero non negativo)
    if (n > 0)
        return n*Fattoriale_Ric (n – 1)
    else return 1
```

Algoritmi ricorsivi (3)

- E' fondamentale assicurarsi che **le funzioni ricorsive abbiano almeno un caso base**, altrimenti la loro esecuzione genererebbe una catena illimitata di chiamate ricorsive che non terminerebbe mai (provocando ben presto la terminazione forzata dell'elaborazione a causa dell'esaurimento delle risorse di calcolo, per ragioni che vedremo fra breve).
- Bisogna quindi essere assolutamente certi che il caso base non possa essere mai "saltato" durante l'esecuzione.
- Nulla vieta di prevedere più di un caso base, basterà assicurarsi che uno tra i casi base sia sempre e comunque incontrato.

Algoritmi ricorsivi (4)

Esempio: la seguente formulazione della funzione che calcola il fattoriale non terminerebbe mai se le venisse passato come valore iniziale un intero negativo:

```
Funzione Fattoriale_Infinita (n: intero)
    if (n = 0)
            return 1
    else return n*Fattoriale_Infinita (n – 1)
```

Algoritmi ricorsivi (5) – ric. sequenziale

Esempio: algoritmo ricorsivo per la ricerca sequenziale su *n* elementi:

- ispezioniamo l'*n*-esimo elemento;
- se non è l'elemento cercato risolviamo il problema ricorsivamente sui primi *(n – 1)* elementi.

```
Funzione Ricerca_sequenziale_ricorsiva(A: vettore; v: intero; n: intero)
    if (A[n] = v) return true
    else if (n = 1) return false
            else return Ricerca_sequenziale_ricorsiva(A, v, n - 1)
```

Esempio: algoritmo ricorsivo per la ricerca binaria su *n* elementi:

```
Funzione Ricerca_binaria_ricorsiva(A: vettore; v:intero; i_min,
    i_max: intero)
    if (i_min > i_max) return null
    m ← i_min+i_max2
    if (A[m] = v)
          return m
    if (A[m] > v)
          return Ricerca_binaria_ricorsiva(A, v, i_min, m - 1)
    else
          return Ricerca_binaria_ricorsiva(A, v, m + 1, i_max)
```

Anche nella formulazione ricorsiva della ricerca binaria ogni nuova chiamata ricorsiva riceve un sottoproblema da risolvere la cui dimensione è circa la metà di quello originario quindi, come nella versione iterativa, si giunge molto rapidamente al caso base.

Da ciò deriva un costo computazionale che, come vedremo, è *O(log n)* nel caso peggiore come per la ricerca binaria iterativa.