Complessità degli algoritmi

- Obiettivi:
  - Calcolare (valutare) la complessità di un algoritmo/programma
  - Confrontare algoritmi risolutivi del medesimo problema

Ricerca minimo in un vettore di N elementi

```
#define  N 100

typedef  int[N] vettore;


int minimo (vettore vet)
{int  i, min;
 for  (min = vet[0], i = 1; i<N ; i ++)
      {if (vet[i]<min)
                   min = vet[i]; }
 return min; }
```

Ricerca esaustiva in un vettore di N elementi

```
int ricerca (vettore vet, int el, int *pos)
{int  i=0;
 int Trovato=0;
 while ((i<N) && (!Trovato))
   { if (el==vet[i])
         {  Trovato=1;
            *pos=i;      }
     else i++;}
 return Trovato;
}
```

Teoria della complessità

- Tra i problemi che ammettono soluzione esistono problemi “facili” e “difficili”.

- **Teoria della complessità (anni ’70), valuta:**
  - complessità intrinseca di un problema;
  - efficienza degli algoritmi risolutivi.

- Qualsiasi programma richiede ***spazio di memoria*** e ***tempo di calcolo***.

Correttezza ed efficienza

- Vogliamo progettare algoritmi che:
  - Producano correttamente il risultato desiderato
  - Siano efficienti in termini di **tempo** di esecuzione ed occupazione di **memoria**

- Ci concentriamo sul ***tempo*** per valutare la ***complessità degli algoritmi*** (**complessità temporale**)
- Valutare la complessità degli algoritmi ci consente di scegliere tra loro quello **più efficiente** (a minor complessità temporale).

Complessità temporale

- Per un algoritmo, è determinata contando il numero di operazioni aritmetiche e logiche, accesso ai file, letture e scritture in memoria, etc.

- **I° ipotesi semplificativa:**
  - Tempo impiegato proporzionale al numero di **operazioni** eseguite (ciascuna a **costo unitario**)

- Non ci si riferisce a una specifica macchina.

Esempio: prodotto matriciale

- Moltiplicazione di due matrici quadrate $n \times n$ di interi:

$$C = A \times B$$

- Per calcolare C[i,j] si eseguono 2n letture, n moltiplicazioni, n-1 addizioni e 1 scrittura.
- Per calcolare C: n^2*(2n letture, n moltiplicazioni, n-1 addizioni ed 1 scrittura):

	$2n^3$	letture
	n^3	*moltiplicazioni*
	$n^2*(n-1)$	*addizioni*
	n^2	*scritture*

$$time_{Alg(C=A\times B)}(n)=2n^3+n^3+n^2*(n-1)+n^{2=}4n^3$$

Dimensione dei dati

- Il tempo impiegato per risolvere un problema dipende sia dall'algoritmo utilizzato sia dalla ***"dimensione" dei dati*** a cui si applica l'algoritmo.

Oggetto di ingresso:	**Dimensione:**
vettore	n elementi
matrice	" "
matrice quadrata	" "
matrice quadrata	n righe
lista	*n elementi*
albero	*n elementi; altezza*

Dimensione dei dati

- La ***complessità*** dell'algoritmo viene espressa in funzione della ***dimensione delle strutture dati*** su cui opera

- $time_{Alg(C=A\times B)}(n)=2n^3+n^3+n^2*(n-1)+n^2=4*n^3$

- $time_{Alg(P)}(n)=2^n$

Ordini di grandezza

n	$\log_2 n$	$n*\log_2 n$	n^2	n^3	2^n
2	1	2	4	8	4
10	3.322	33.22	10^2	10^3	$> 10^3$
10^2	6.644	664.4	10^4	10^6	$>> 10^{25}$
10^3	9.966	9996.0	10^6	10^9	$>> 10^{250}$
10^4	13.287	1328.7	10^8	10^{12}	$>> 10^{2500}$

- Con un elaboratore che esegue 10^3 operazioni al sec., l'algoritmo risolutivo del problema P (che ha complessità 2^n) con ingresso di dimensione:
  - 10 impiega 1 sec
  - 20 " 1000 sec (17 min)
  - 30 " 10^6 sec (>10 giorni)
  - 40 " 10^9 sec (>>10 anni)

Comportamento asintotico

- Individuare con esattezza $time_A(n)$ è spesso molto difficile.

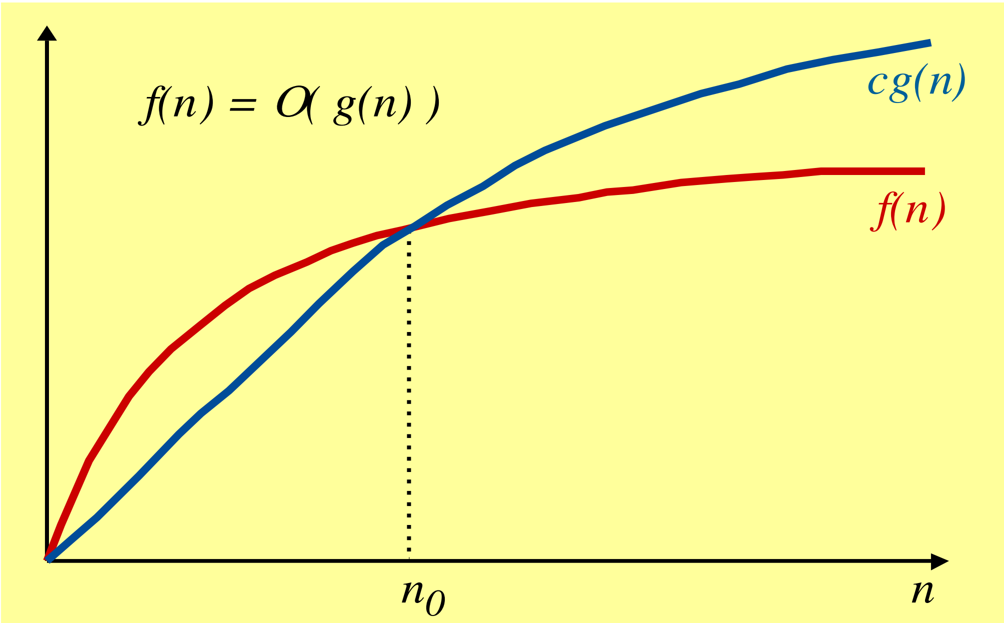
- **II° ipotesi semplificativa:**
  - E'sufficiente stabilire il comportamento asintotico della funzione quando le dimensioni dell'ingresso tendono ad infinito (***comportamento asintotico dell'algoritmo***).

- Si usa a questo scopo la notazione O()

Notazione asintotica O( )

- Un algoritmo (programma) ha costo: **f(n)=O(g(n))** (o complessità O(g(n))) se esistono opportune costanti c, n_0, tali che:

f(n) < c*g(n) **per ogni n> n_0.**



(C. Demetrescu, I. Finocchi, G. F. Italiano, "Algoritmi e strutture dati", seconda edizione, McGraw-Hill)

- O(g(n)), limite superiore al comportamento asintotico di una funzione.

Esempi

- $3*n^2+4*n+3 = O(n^2)$

perchè: $3*n^2+4*n+3 <= 4*n^2$ per ogni $n>3$.

- Prodotto matriciale:

$$time_{Alg(C=AxB)}(n)=4*n^3 = O(n^3)$$

- Ricerca del massimo in un vettore:

$$time_{search}(n)=n = O(n)$$

Ordini di complessità

- Attraverso la notazione O( ), gli algoritmi vengono divisi in ***classi di equivalenza***, ponendo nella medesima classe tutti quelli la cui ***complessità asintotica è dello stesso ordine di grandezza***.

- Si hanno così algoritmi (funzioni) di complessità asintotica di ordine:
  - Costante 1, ...
  - Sotto-lineare $\log n$, n^k con $k<1$
  - Lineare n
  - Polinomiale $n*\log n$, n^2, n^3, ... n^k con $k>1$
  - Esponenziale c^n, n^n, ...

Algoritmi e tempi tipici

Moltissimi algoritmi

Tre tempi tipici: $O(n)$, $O(n \log_2 n)$, $O(n^2)$

n	10	100	1000	10^6	10^9
$n \log_2 n$	~ 33	~ 665	~ 10^4	~ $2 \cdot 10^7$	~ $3 \cdot 10^{10}$
n^2	100	10^4	10^6	10^{12}	10^{18}

Una lecita domanda … (fin qui)

- A cosa serve la teoria della complessità?
  - Serve a valutare quanto “difficile” è un problema
  - Serve a valutare le risorse (spazio e tempo) necessarie per lo svolgimento automatico di un algoritmo (programma)
  - Serve per confrontare algoritmi risolutivi dello stesso problema
  - …
  - Conoscerla ci aiuta a individuare algoritmi (programmi) più efficienti

Algoritmo migliore

- Dato un problema P e due algoritmi A1 e A2 che lo risolvono siamo interessati a determinare quale ha complessità minore (è "il migliore").

- Dato un problema P e due algoritmi A e B che lo risolvono con complessità timeA e timeB, diciamo che A è ***migliore*** di B nel risolvere P se:
  - (1) timeA=O(timeB)
  - (2) timeB non è O(timeA)

n	$\log_2 n$	// ricerca
$n \log_2 n$	n^2	// sorting

Istruzione dominante

- Il tempo t(n) dell'algoritmo è dominato dal numero di volte d(n) in cui è eseguita questa istruzione (n dimensione input): **$t(n) < a\ d(n) + b$**
- E' eseguita un numero di volte proporzionale al costo dell'algoritmo.
- **III° ipotesi semplificativa:**
  - Se esiste un'istruzione dominante, la complessità dell'algoritmo è **$O(d(n))$**
- Come identificare le istruzioni dominanti?
  - **Modello basato sui confronti**

Modello basato su confronti

- **IV° ipotesi semplificativa:** Si conteggiano le sole operazioni di confronto e si trascurano le altre operazioni primitive
- Adottato per molti algoritmi che lavorano su strutture di dato
- Sufficientemente generale per catturare le proprietà dei principali algoritmi (su strutture di dato)

ALGORITMI DI RICERCA SU VETTORI

```
#include <stdio.h>
#define N 15
typedef int vettore[N];

void main (void )
{int i;
 vettore a;
 printf ("Scrivi %d numeri interi\n", N);
 for (i = 0; i < N; i++)
          { scanf ("%d", &a[i]); }
 printf ("Il minimo vale %d e il massimo è %d\n",
            minimo(a), massimo(a));
}
```

```
int minimo (vettore vet)
{int  i, min;
 for (min = vet[0], i = 1; i < N; i ++)
     {if (vet[i]<min)              /* istr. dom. */
                  min = vet[i]; }
 return min; }

int massimo (vettore vet)
{int  i, max;
 for (max = vet[0], i = 1; i < N; i ++)
     {if (vet[i]>max)          /* istr. dominante*/
                  max=vet[i];}
  return max; }
```

- Per la ricerca sia del minimo sia del massimo, *l'istruzione dominante* viene eseguita N-1 volte.
- Costo O(N), se N è la dimensione del vettore.

Cambia qualcosa con istruzione while?

```
int minimo (vettore vet)
{int  i, min;
min = vet[0]; i = 1;
while (i < N)
      {if (vet[i]<min)                /* istr. dom. */
                  min = vet[i];
       i ++)
}
 return min; }
```

- Per la ricerca sia del minimo sia del massimo, *l'istruzione dominante* viene sempre eseguita N-1 volte.
- Costo O(N), se N è la dimensione del vettore.

Caso peggiore, migliore e medio

- Molto spesso il costo dell'esecuzione di un programma (di un algoritmo) dipende non solo dalla dimensione dell'ingresso, ma anche dai particolari valori dei dati in ingresso.

- È possibile distinguere diversi casi: caso migliore, caso peggiore, caso medio.

- Di solito la complessità viene valutata nel ***caso peggiore*** (e talvolta nel ***caso medio***).

ALGORITMI DI RICERCA SU VETTORI

```
void main (void )
{int i, p;
 vettore a;
    . . . /* lettura elementi di a */
 printf ("Scrivi un intero\n");
 scanf ("%d", &i);
 if ( ricerca(a,i,&p) )
       printf("\n Trovato in posizione %d\n", p);
 else printf("\n Non trovato\n");
}
```

Ricerca esaustiva in un vettore di N elementi

```
int ricerca (vettore vet, int el, int *pos)
{int  i=0;
 int Trovato=0;
 while (i<N)                              /* (1) */
   { if (el==vet[i])                       /* (2) */
          {  Trovato=1;
             *pos=i;      }
     i++;}
 return Trovato;
}
```

- (1) e (2) istruzioni dominanti
- eseguite N+1 e N volte (nel caso peggiore)

Ricerca esaustiva in un vettore di N elementi

```
int ricerca (vettore vet, int el, int *pos)
{int  i=0;
 int Trovato=0;
 while ((i<N)&&(!Trovato))                    /* (1) */
   { if (el==vet[i])                           /* (2) */
         {  Trovato=1;
            *pos=i;       }
      else i++;}
 return Trovato;
}
```

- (1) e (2) istruzioni dominanti
- eseguite N+1 e N volte (nel caso peggiore)

2	6	1	5	7	8	3	10

Ricerca esaustiva (o sequenziale)

- Per la ***ricerca sequenziale*** in un vettore il costo dipende dalla posizione dell'elemento cercato.

- ***Caso migliore***, l'elemento è il primo del vettore (1 confronto)

- ***Caso peggiore***, l'elemento è l'ultimo o non è presente: l'istruzione dominante è eseguita N volte (N dimensione del vettore). Il costo è ***lineare***, O(N)

- ***Caso medio***, ciascun elemento sia equiprobabile

$$\sum_{(i=1..N)} \text{Prob}(\text{el}(i))*i = \sum_{(i=1..N)} (1/N)*i = (N+1)/2$$

È sempre O(N)

Ricerca in array

- Se l'array non è ordinato → ricerca esaustiva (o sequenziale)
- Se l'array è ordinato → ricerca binaria

- La tecnica di ***ricerca binaria***, rispetto alla ricerca esaustiva, consente di ***eliminare ad ogni passo metà degli elementi del vettore***

- Nota: conviene ordinare un array per usare la ricerca binaria?
  - ***Dipende*** → *si vedrà poi in quali condizioni e perché ...*

Ricerca di un elemento

Sapendo che il vettore è ***ordinato*** (esiste una relazione d'ordine totale sul dominio degli elementi), la ricerca può essere ottimizzata

- ***Vettore ordinato in senso non decrescente:***

2	3	5	5	7	8	10	11

`se i<j` si ha **`V[i]<=V[j]`**

- ***Vettore ordinato in senso crescente:***

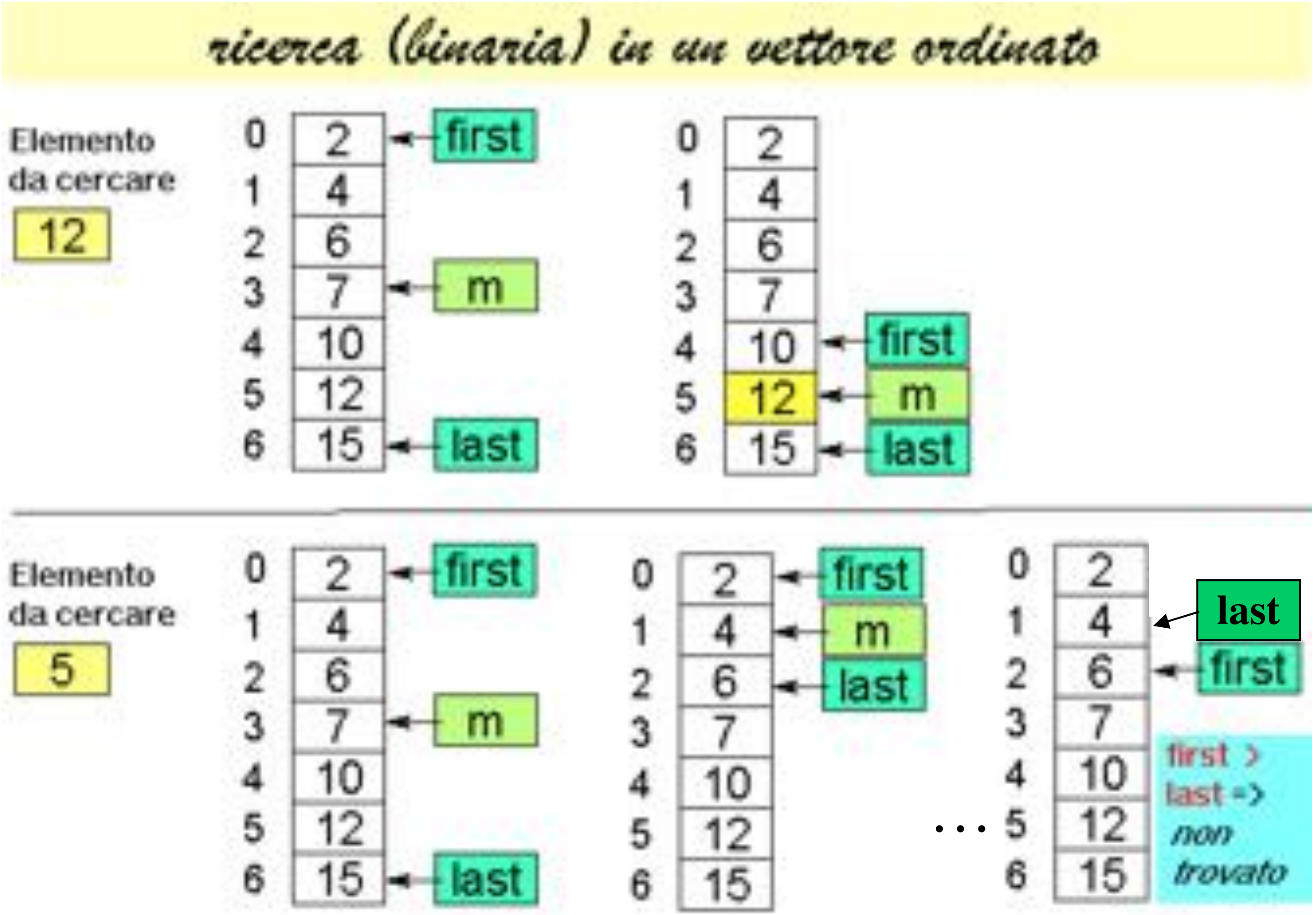
2	3	5	6	7	8	10	11

`se i<j` si ha **`V[i]<V[j]`**

In modo analogo si definiscono l'ordinamento in senso ***non crescente*** e ***decrescente***

RICERCA BINARIA

Esempio

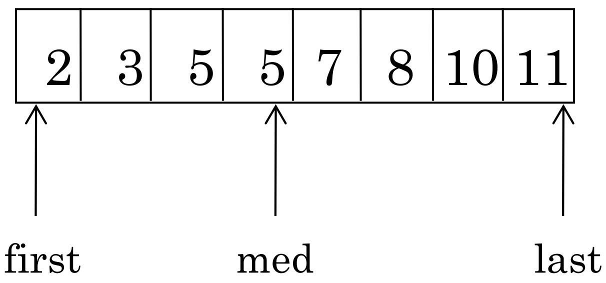


Ricerca binaria in un vettore (da indice *first* a *last*)

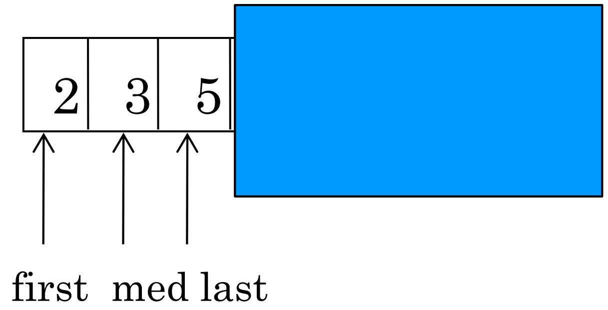
- Vettore con indici da *first* a *last*; indice mediano *med=(first+last)/2*
- Si confronta l'elemento cercato **`el`** con quello mediano del vettore, **`V[`*`med`*`]`**
- Se **`el==V[med]`**, fine della ricerca (**`trovato=true`**)
- Altrimenti, se il vettore ha almeno una componente (**`first<=last`**):
  - se **`el<V[med]`**, ripeti la ricerca nella prima metà del vettore (indici da **`first`** a **`med-1`**)
  - se **`el>V[med]`**, ripeti la ricerca nella seconda metà del vettore (indici da **`med+1`** a **`last`**)

Ricerca binaria: esempio

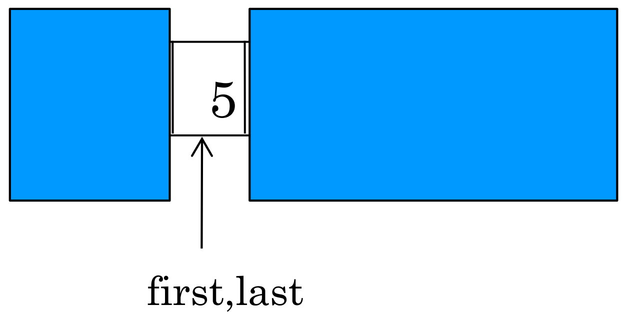
Si cerchi il valore 4




```
med= (first+last)/ 2
el<V[med]
```




```
el>V[med]
```



Vettore mono-dimensionale

```
/* ricerca binaria su un vettore di interi   */
typedef enum {falso, vero} boolean;

boolean ricerca_bin (vettore vet, int el, int *pos)
{int  first=0,
      last=N-1,
      med=(first+last)/2 ;
boolean Trovato=falso;
 while ((first<=last)&&(!Trovato))  /* istr. dom. */
   { if (el==vet[med])
              {Trovato=vero; *pos=med;}
     else
        if (el < vet[med])  last=med-1;
        else   first=med+1;
     med = (first + last) / 2; }
 return  Trovato;
}
```

Chiamata:

```
if (ricerca_bin(a,i,&p))
  printf("\nTrovato in posizione
              %d\n", p);
else printf("\nNon trovato\n");
```

Ricerca binaria: complessità

- Come per la ricerca sequenziale, dipende dalla posizione dell'elemento cercato.

- ***Caso migliore***, l'elemento cercato è quello mediano nel vettore (1 confronto)

- ***Caso peggiore***, l'elemento cercato è l'ultimo o non è presente nel vettore. Il ciclo **`while`** è ripetuto finché ci si riduce ad un vettore mono-dimensionale (**`first==last`**).

- Ad ogni passo di iterazione la dimensione del vettore dimezza, si hanno al più k passi (con k finito e proporzionale a $\log_2 N$):

Ricerca binaria ($N=2^k$): caso peggiore

Passo	n. elem vettore	n. confronti
1	N	1
2	N/2	1
3	N/4	1
…		
k	$N/2^{(k-1)}$	1
k+1	1	1

$$\sum_{(i=1..k+1)} 1 = k+1 = \log_2 N + 1$$

$O(\log_2 N)$

ESERCIZIO (in Laboratorio)

- È dato un vettore di dimensione N+k contenente N numeri interi (N può essere anche 0), ordinati in senso non decrescente

- Ricevendo uno alla volta k interi, li si inserisca nel vettore, mantenendo l'ordinamento del vettore ad ogni passo di inserimento

- Determinare le condizioni che generano il maggior numero di confronti tra elementi, in funzione di N e k